HRVATSKA AKADEMSKA I ISTRAŽIVAČKA MREŽA
CROATIAN ACADEMIC AND RESEARCH NETWORK

Zaobilaženje mehanizma zaštite stoga na Windows Server 2003 platformi

CCERT-PUBDOC-2003-09-40

CARNet CERT u suradnji s LS&S

Sigurnosni problemi u računalnim programima i operativnim sustavima područje je na kojem CARNet CERT kontinuirano radi.

Rezultat toga rada ovaj je dokument koji je nastao suradnjom CARNet CERT-a i LS&S-a, a za koji se nadamo se da će Vam koristiti u poboljšanju sigurnosti Vašeg sustava.

CARNet CERT, www.cert.hr - nacionalno središte za **sigurnost računalnih mreža** i sustava.

LS&S, www.lss.hr - laboratorij za sustave i signale pri Zavodu za elektroničke sustave i obradbu informacija Fakulteta elektrotehnike i računarstva Sveučilišta u Zagrebu.

Sadržaj

1. Uvod

Microsoft svoje nove proizvode nastoji isporučiti što sigurnijim, što je posljedica nove politike tvrtke koja je zaokružena u sloganu "*Trustworthy Computing*". Bez obzira na to, čak i u najnovijim operacijskim sustavima, kao što je Microsoft Windows Server 2003, i dalje se pronalaze razni sigurnosni nedostaci.
Velik dio tih sigurnosnih nedostataka temelji se na iskorištavanju mogućnosti prepisivanja spremnika (engl. *buffer overflow*). Međutim, unutar Windows Server 2003, kao jedna od sigurnosnih inovacija, implementiran je mehanizam zaštite od napada prepisivanjem spremnika. Nažalost, iako predstavlja pomak u pozitivnom smjeru, pokazuje se da i ovaj mehanizam može biti zaobiđen.

2. Zaštita stoga unutar Windows Server 2003

Windows Server 2003 donosi mehanizam zaštite stoga koji je integriran u prevoditelj i koji bi trebao predstavljati zaštitu od napada prepisivanjem spremnika. Sam koncept nije revolucionaran, i već je ranije implementiran u nekim od postojećih proizvoda na tržištu (npr. StackGuard, http://immunix.org/).
Osnovna ideja jest da se na stogu, ispred povratne adrese funkcije, postavi sigurnosni kolačić (engl. *cookie*). U tom slučaju, ukoliko dođe do prepisivanja lokalnog spremnika u odnosu na određenu funkciju, osim prepisivanja pohranjene povratne adrese, prepisat će se i kolačić. Prije povratka iz funkcije provjerava se i integritet kolačića. To se čini usporedbom s referentnim kolačićem pohranjenim u `.data` dijelu modula, unutar kojeg se funkcija nalazi. Ukoliko kolačići nisu identični pretpostavlja se da je došlo do prepisivanja spremnika i proces se zaustavlja.
Ovakav sigurnosni mehanizam osiguran je kroz Visual Studio .NET korištenjem GS zastavice koja je predefinirano uključena.

2.1. Način generiranja sigurnosnog kolačića

Prilikom učitavanja modula, kao dio *start-up* rutine generira se i sigurnosni kolačić. Kolačić se generira na slučajni način, tako da je predviđanje vrijednosti koju će poprimiti vrlo teško, pogotovo ukoliko napadač ima samo jednu mogućnost izvršavanja napada. U osnovi, kolačić se generira izvođenjem višestrukih XOR operacija nad povratnim vrijednostima nekoliko funkcija. Programski kod zadužen za generiranje kolačića izgleda na sljedeći način:

```
#include <stdio.h>
#include <windows.h>
int main()
{
FILETIME ft;
unsigned int Cookie=0;
unsigned int tmp=0;
unsigned int *ptr=0;
LARGE_INTEGER perfcount;
GetSystemTimeAsFileTime(&ft);
Cookie = ft.dwHighDateTime ^ ft.dwLowDateTime;
Cookie = Cookie ^ GetCurrentProcessId();
Cookie = Cookie ^ GetCurrentThreadId();
Cookie = Cookie ^ GetTickCount();
QueryPerformanceCounter(&perfcount);
ptr = (unsigned int)&perfcount;
tmp = *(ptr+1) ^ *ptr;
Cookie = Cookie ^ tmp;
printf("Cookie: %.8X\n",Cookie);
return 0;
}
```

Kolačić je prikazan kao nepredznačeni cijeli broj (engl. *unsigned int*), a nakon što je generiran pohranjuje se u `.data` (podatkovnom) dijelu modula. Taj dio memorije nije zaštićen od pisanja što

je, kako će biti pokazano u nastavku, jedna od tehnika napada koja iskorištava način da se referentni kolačić prepiše s poznatom vrijednošću.

2.2. Usporedba kolačića

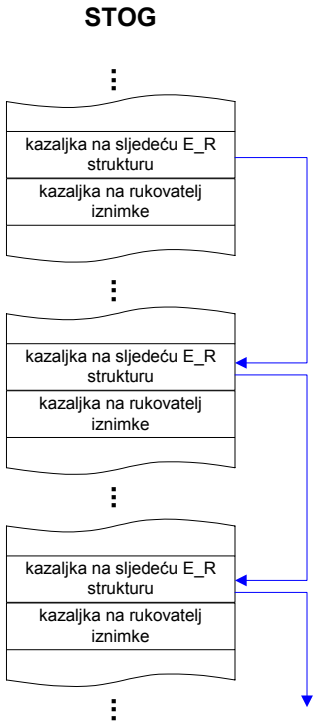
Mehanizam zaštite stoga aktivira se ukoliko dođe do prepisivanja lokalnog spremnika, što također podrazumijeva i prepisivanje sigurnosnog kolačića. Prilikom provjere kolačića ustanovit će se da je došlo do prepisivanja, a programski kôd će nakon toga provjeriti da li je eventualno definiran sigurnosni rukovatelj (engl. *security handler*). Ovaj rukovatelj postoji tako da bi programeri mogli poduzeti odgovarajuće akcije (npr. zabilježiti stanje aplikacije prije termininiranja procesa). Kazaljka na rukovatelj također je pohranjena u `.data` dijelu programskog modula. Definicija rukovatelja nije nužna, štoviše, sa sigurnosnog stajališta možda je i uputno izbjegavanje njegovog korištenja, pošto nakon prepisivanja spremnika može doći do kompromitacije i tog dijela kôda. Najjednostavniji i najsigurniji postupak je bezuvjetno termininiranje procesa. Čak i ukoliko sigurnosni rukovatelj nije definiran, još uvijek postoji dio kôda koji se izvršava kao dio procesa.

Predefinirani postupak prilikom detekcije nepodudaranja kolačića jest postavljanje `UnhandledExceptionFilter()` vrijednosti na `0x00000000` i pozivanje `UnhandledExceptionFilter()` funkcije. Na taj način započeto je termininiranje procesa. `UnhandledExceptionFilter()` funkcija obavlja nekoliko zadataka prije nego što se proces konačno terminira. Između ostalog, funkcija učitava `faultrep.dll` biblioteku i poziva `ReportFault()` funkciju (često prikazana poruka "*Report this fault to Microsoft*" posljedica je rada ove funkcije). Kako je ranije spomenuto, svako izvršavanje programskog kôda nakon prepisivanja spremnika potencijalno je opasno, što će kasnije biti i pokazano na primjeru, pošto se `UnhandledExceptionFilter()` funkcija može iskoristiti za dobivanje kontrole nad procesom kod kojeg je došlo do prepisivanja spremnika.

2.3. Rukovanje iznimkama

Rukovanje iznimkama osigurava robusnost aplikacija i integralni je dio Windows operacijskih sustava. Kada se pojavi pogreška kao što je nepravilni pristup memoriji ili dijeljenje s nulom, moguće je definirati rukovatelj iznimke koji će u takvim situacijama omogućiti povratak procesa u normalno stanje i nastavak izvršavanja. Čak i ukoliko programer nije definirao nikakav mehanizam rukovanja iznimkama, svaka nit (engl. *thread*) svakog procesa ima barem jedan rukovatelj koji se postavlja prilikom inicijalizacije niti. Podaci o rukovateljima iznimaka pohranjeni su na stogu u `EXCEPTION_REGISTRATION` strukturi, a kazaljka na prvu `EXCEPTION_REGISTRATION` strukturu pohranjena je u bloku koji definira okruženje niti (engl. *Thread Environment Block*).

Kako je prikazano na slici (*Slika 1*), `EXCEPTION_REGISTRATION` struktura sastoji se od dva elementa: kazaljke na sljedeću `EXCEPTION_REGISTRATION` strukturu i kazaljke koja pokazuje na rukovatelj iznimke. Prilikom prepisivanja stoga na Windows platformama, jedna od mogućnosti koju napadač može iskoristiti za dobivanje kontrole nad procesom je prepisivanje `EXCEPTION_REGISTRATION` strukture. Napadač to može napraviti postavljanjem kazaljke rukovatelja na dio kôda ili instrukciju koja će vratiti izvršavanje nazad u granice spremnika.



Slika 1: Mehanizam rukovanja iznimkama

3. Mehanizmi napada

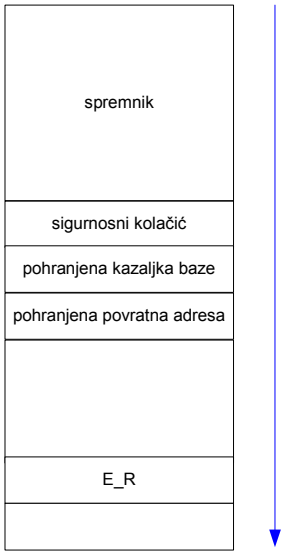
Rukovanje iznimkama i mogućnost zlouporabe, kako je u prethodnom poglavlju opisano, predstavlja potencijalni sigurnosni nedostatak, tako da je u Windows Server 2003 mehanizam strukturiranog rukovanja iznimkama promijenjen u cilju zaštite od takvih napada. Rukovatelji iznimki se registriraju i adrese svih rukovatelja u modulu se pohranjuju u konfiguracijski direktorij modula (engl. *Load Configuration Directory*). Kada se pojavi iznimka, prije nego što se rukovatelj izvrši, njegova adresa se provjerava u odnosu na listu registriranih rukovatelja, a rukovatelj se izvodi samo ukoliko je pronađena odgovarajuća adresa. Međutim, ukoliko je adresa rukovatelja koja se nalazi u `EXCEPTION_REGISTRATION` strukturi izvan raspona adresa učitanog modula, rukovatelj se također izvodi. Posljedica toga je da ukoliko se adresa rukovatelja nalazi na stogu, rukovatelj neće biti izvršen. Na ovaj način moguće se zaštititi od napada gdje je kazaljka rukovatelja iznimke direktno prepisana adresom sa stoga. Ukoliko je adresa rukovatelja negdje unutar gomile (engl. *heap*), rukovatelj će ipak biti izvršen.

Ovaj mehanizam otvara napadaču dvije mogućnosti za napade, isto kao i eventualnu treću, no ta treća mogućnost je mnogo teža i ovisi o samom kôdu, odnosno procesu. Prilikom prepisivanja lokalnog spremnika, isto kao i prilikom prepisivanja sigurnosnog kolačića i pohranjene povratne adrese, napadač prepisuje `EXCEPTION_REGISTRATION` strukturu, da bi nakon toga pokušao doći u kontrolu procesa postavljanjem kazaljke rukovatelja na već registrirani rukovatelj ili prepisivanjem kazaljke tako da ona pokazuje na rukovatelj izvan adresnog prostora učitanog modula koji sadrži dio kôda ili instrukciju koja omogućava napadaču povratak u spremnik. Posljednja i najsloženija mogućnost napada je učitavanje spremnika na gomilu, zajedno sa malicioznim kôdom i prepisivanje kazaljke rukovatelja s adresom gomile.

3.1. Napad iskorištavanjem strukturiranog rukovanja iznimkama

Da bi se mehanizam strukturiranog rukovanja iznimkama mogao iskoristiti za zaobilaženje zaštite stoga potrebno je prije provjere sigurnosnog kolačića generirati iznimku. To je moguće napraviti na dva načina; jedan način je moguće kontrolirati, dok je drugi ovisan o kôdu napadnute funkcije. U

ovom drugom slučaju, prepisivanjem ostalih podataka, npr. parametara koji su potisnuti na stog i njihovim referenciranjem prije provjere sigurnosnog kolačića je moguće generirati iznimku postavljanjem podataka na vrijednost koja će uzrokovati iznimku. Ukoliko je napadnuta funkcija napisana tako da omogućava ovakav napad, moguće je pokušati generirati vlastitu iznimku. Iznimku je moguće generirati pokušajem pisanja izvan granice stoga (*Slika 2*).



Slika 2: Prepisivanje kazaljke rukovatelja iznimke adresom izvan adresnog raspona učitanog modula

Valja imati na umu da se prije izvršavanja rukovatelja iznimke njegova adresa provjerava u odnosu na listu registriranih rukovatelja. Ukoliko je adresa rukovatelja iznimke izvan adresnog prostora, rukovatelj će se ipak izvršiti. Zbog takvog načina rada napadaču je dovoljno pronaći instrukciju ili blok instrukcija koji će izvršavanje vratiti ponovno na korisnički definirane podatke na adresi izvan raspona modula. Ako se provjeri stanje stoga i registara prilikom poziva rukovatelja iznimke, može se uočiti da niti jedan od registara ne sadrži adresu unutar korisnički definiranih podataka. Ukoliko bi to bilo moguće, napadač bi samo trebao pronaći `CALL REGISTER` ili `JMP REGISTER` instrukcije, no niti jedan od registara ne pokazuje na područje korisno napadaču. Pregledom stanja stoga može se uočiti da je prilikom poziva rukovatelja iznimke prepisano više kazaljki na `EXCEPTION_REGISTRATION` strukturu (*Tablica 1*).

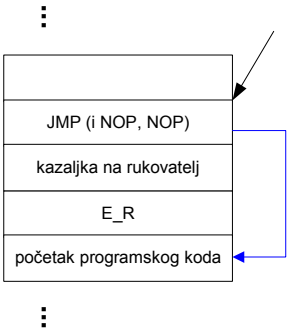
ESP	+8	+14	+1C	+2C	+44	+50
EBP	+0C	+24	+30	-04	-0C	-18

Tablica 1: Sadržaj ESP i EBP prilikom poziva rukovatelja iznimke

Ukoliko napadač može pronaći instrukciju koja izvršava sljedeći niz:

```
CALL DWORD PTR [ESP+NN]
CALL DWORD PTR [EBP+NN]
CALL DWORD PTR [EBP-NN]
JMP DWORD PTR [ESP+NN]
JMP DWORD PTR [EBP+NN]
JMP DWORD PTR [EBP-NN]
```

gdje je `NN` jedan od posmaka pronađenih na stogu (*Tablica 1*), na adresi izvan adresnog prostora učitanog modula, tada će ona biti izvršena prilikom događaja iznimke. Na taj način tijek izvođenja instrukcija će biti prebačen na kontroliranu strukturu za obradu iznimki. Kako su prva 32 bita ove strukture kazaljka na sljedeću strukturu, napadač može učitati kratki kôd koji će izvesti skok preko kazaljke na rukovatelj (*Slika 3*).



Slika 3: Skok na umetnuti programski kôd

Alternativa jest pronaći `POP REG, POP REG, RET` blok instrukcija izvan adresnog prostora učitanog modula. Taj način funkcionira pošto se može pronaći kazaljka na `EXCEPTION_REGISTRATION` strukturu na poziciji `ESP+8`. Dva podizanja sa stoga će postaviti kazaljku na `ESP`, tako da se prilikom izvođenja `RET` instrukcije izvođenje vraća nazad u strukturu. U tom trenutku ponovno je potreban kratki skok. Adresni prostor ranjivog procesa u potrazi za takvim blokom instrukcija moguće je pretražiti umetanjem kratkog kôda koji će učitati dinamičku biblioteku koja sadrži kôd za pretraživanje (Dodatak A). Izlaz iz programskog kôda pokazuje tri mogućnosti gdje se odgovarajuće instrukcije mogu pronaći izvan adresnog prostora učitanog modula. Kao prvo, tu je `CALL DWORD PTR[EBP+30h]` na adresi `0x001B0B0B` te dva bloka `POP REG, POP REG, RET` instrukcija na `0x7FFC0AC5` i `0x7FFC0AFC`.

Sve te tri adrese nalaze se izvan adresnog prostora učitanog modula. `CALL DWORD PTR[EBP+30h]` ustvari nije instrukcija, već samo odgovarajući okteti koji čine ovu instrukciju `0xFF5530`. Analizom memorije procesa može se vidjeti da se odgovarajući okteti nalaze unutar adresnog prostora mapirane podatkovne datoteke `UNICODE.NLS`. Ta datoteka mapira se u nekim procesima, dok u drugima to pak ne mora biti slučaj. Bazna adresa `UNICODE.NLS` datoteke identična je za svaku instancu istog procesa, ali može varirati između različitih procesa. Npr., `UNICODE.NLS` će uvijek biti mapiran na `0x00260000` za danu konzolnu aplikaciju, dok će za `svchost.exe` ta adresa biti `0x001A0000`. Instrukcija `CALL DWORD PTR[EBP+30h]` može biti pronađena na posmaku od `0x10B0B` okteta od bazne adrese.

Proces	Bazna adresa	Broj
SERVICES.EXE	0x001A0000	-
LSASS.EXE	0x001A0000	-
SVCHOST.EXE	0x001A0000	svih 8 instanci
SPOOLSV.EXE	0x001A0000	-
MSDTC.EXE	0x001A0000	-
DFSSVC.EXE	0x001A0000	-
WMIPRVSE.EXE	0x001A0000	obje instance
W3WP.EXE	0x001A0000	-
CSRSS.EXE	0x00270000	-
SMSS.EXE	UNICODE.NLS nije mapiran	-
INETINFO.EXE	UNICODE.NLS nije mapiran	-
CIDAEMON.EXE	UNICODE.NLS nije mapiran	-
CISVC.EXE	UNICODE.NLS nije mapiran	-

Tablica 2: Neki servisi koji mapiraju UNICODE.NLS

Tablica 2 prikazuje neke od servisa koji su uobičajeni na Windows 2003 Server poslužitelju. Kako može uočiti, većina servisa mapira `UNICODE.NLS` na adresi `0x001A0000` tako da se `CALL DWORD PTR[EBP+30h]` instrukcija može locirati na `0x001B0B0B`.

Preostale dvije adrese koje sadrže `POP, POP, RET` blok instrukcija donekle su različite. Iako su te adrese identične za svaki proces, čak i nakon restarta, na svim sustavima se nalaze na istoj lokaciji.

Lokacija tih adresa vrlo vjerojatno je ovisna o sklopovlju, no zbog te nestabilnosti one se uglavnom mogu koristiti samo za lokalne napade, osim ukoliko napadač ne poznaje točnu sklopovsku specifikaciju udaljenog računala, što uglavnom nije slučaj. To znači da se napadač, nakon što je provjerio adresu `UNICODE.NLS` ranjivog procesa na vlastitom računalu, mora osloniti na `CALL DWORD PTR[EBP+30h]` instrukciju. Pošto je vjerojatnije da ta adresa sadrži `NULL` vrijednost umjesto skoka naprijed, napadač treba planirati skok u nazad. `NULL` vrijednost može predstavljati i problem u nekim slučajevima. Iznimka se sigurno događa ukoliko se dogodi prepisivanje s pokušajem pisanja izvan granica stoga. No, s `NULL` vrijednošću generiranje iznimke na taj način nije moguće, osim ukoliko prepisivanje ne postoji u samoj Unicode datoteci (kao npr. unutar DCOM-a). Ukoliko to nije slučaj, napadač se mora nadati da će doći do kršenja prava pristupa kod samog kôda ili mora pronaći instrukciju na adresi koja ne sadrži `NULL` vrijednost.
Ovakav način upravljanja iznimkama nije siguran. Nije sigurno pozvati rukovatelj iznimke izvan adresnog prostora učitanog modula, pošto u tom slučaju postoji velik prostor koji napadač može iskoristiti da bi zaobišao zaštitu stoga. Možda bi najbolje rješenje bilo premještanje kôda rukovatelja iznimki izvan adresnog prostora učitanog modula (npr. na gomilu).

3.2. Napad iskorištavanjem postojećeg registriranog rukovatelja iznimki

Ukoliko se pretpostavi da ne postoje instrukcije izvan adresnog prostora modula koje bi zadovoljavale ranije opisani slučaj, moguće je primijeniti drugu tehniku zaobilaženja zaštite stoga. Naime, postoji registrirani rukovatelj u konfiguracijskom direktoriju `NTDLL.DLL` s adresom `0x77F45A34`. Disasembliranjem tog rukovatelja dolazi se do sljedećeg kôda:

```
77F45A3F mov ebx,dword ptr [ebp+0Ch]
..
77F45A61 mov esi,dword ptr [ebx+0Ch]
77F45A64 mov edi,dword ptr [ebx+8]
..
77F45A75 lea ecx,[esi+esi*2]
77F45A78 mov eax,dword ptr [edi+ecx*4+4]
..
77F45A8F call eax
```

Na adresi `0x77F45A34` vrijednost koja pokazuje na `EBP+0Ch` se sprema na `EBX`. Ta vrijednost je kazaljka na kontroliranu `EXCEPTION_REGISTRATION` strukturu. `DWORD` vrijednost koja pokazuje na `0x0C` okteta nakon `EBX` se posprema u `ESI`. Prepisivanjem `EXCEPTION_REGISTRATION` strukture napadač može kontrolirati `DWORD` vrijednost, a samim time i `ESI` registar. Nakon toga se `DWORD` vrijednost koja pokazuje na `0x08` okteta nakon `EBX` posprema na `EDI`. Ponovno, napadač ima kontrolu nad time. Efektivna adresa `ESI+ESI*2` (`ESI*3`) se učitava u `ECX`. Pošto napadač kontrolira `ESI`, može podesiti i vrijednost koja se učitava u `ECX`. Nakon toga se adresa na koju pokazuje `EDI`, dodana na `ECX*4+4`, posprema na `EAX` koji se zatim poziva. Pošto napadač preko `ESI` registra kompletno kontrolira koje vrijednosti se pohranjuju na `EDI` i `ECX`, može kontrolirati vrijednost koja će se pohraniti na `EAX` i na taj način usmjeriti proces da izvrši njegov kôd. Jedini problem je pronalaženje adrese koja sadrži kazaljku na napadačev kôd. Napadač mora osigurati da vrijednost EDI+ECX*4+4 odgovara toj adresi, tako da se kazaljka na napadački kôd pohrani na EAX i zatim pozove. Da bi to bilo uspješno potrebna je kazaljka na kontroliranu `EXCEPTION_REGISTRATION` strukturu. Napadač može do te vrijednosti doći preko stoga, pošto su `EXCEPTION_REGISTRATION` strukture međusobno povezane.
Uz pretpostavku da postoji kazaljka na kontroliranu `EXCEPTION_REGISTRATION` strukturu na adresi `0x005CF3E4`, ukoliko napadač postavi `0x0C` okteta nakon `EXCEPTION_REGISTRATION` strukture na `0x40001554` (što se pohranjuje na `ESI`) i `0x08` okteta nakon strukture na `0x005BF3F0` (što se pohranjuje na `EDI`), nakon što se izvrše multiplikacije i dodavanja preostaje vrijednost `0x005CF3E4`. `DWORD` vrijednost na koju pokazuje ta adresa pohranjuje se na `EAX` i zatim poziva. Nakon poziva EAX izvršavanje se nastavlja unutar `EXCEPTION_REGISTRATION` strukture, na mjestu gdje je trebala biti kazaljka na sljedeću `EXCEPTION_REGISTRATION` strukturu. Ukoliko

napadač na to mjesto može umetnuti kôd koji će izvršiti `JMP` instrukciju za 14 okteta u odnosu na trenutnu lokaciju može dovesti točku izvršavanja upravo na željeno mjesto.
Lokacija ove kazaljke mora biti stabilna, pošto u suprotnom slučaju ovakav napad neće biti funkcionalan.
Dok god većina rukovatelja izvodi skok na `__except_handler3` u `msvcrt.dll` ili ga jednostavno kopira, rukovatelj na adresi `0x77F45A34` nije jedini koji se može iskoristiti na ranije opisani način.

3.3. Napadi korištenjem metoda ovisnih o kôdu

Ovo poglavlje opisuje neke od metoda koje se mogu koristiti kada sam kôd ranjive funkcije to dozvoljava.
Kao primjer će se koristiti sljedeća funkcija:

```
int foo(char **bar)
{
char *ptr = 0;
char buffer[200]="";
strcpy(buffer,bar);
..
..
*bar = ptr;
return 0;
}
```

Prije povratka iz funkcije kazaljka `bar` se postavlja da pokazuje na `ptr`. Asemblirani kôd izgleda na sljedeći način:

```
MOV ECX, DWORD PTR [EBP+8] // **bar
MOV EDX, DWORD PTR [EBP-8] // ptr
MOV DWORD PTR [ECX],EDX
```

Kao što se može vidjeti adresa kazaljke `bar` se pohranjuje u `ECX`, dok se vrijednost `ptr` pohranjuje u `EDX`, koji se zatim pohranjuje na adresu na koju pokazuje `ECX`. U izvornom kôdu se može uočiti da postoji nesiguran poziv `strcpy()` funkcije što može rezultirati prepisivanjem lokalnog spremnika. Ukoliko napadač prepiše taj spremnik, prepisat će i sigurnosni kolačić, zatim i pohranjenu povratnu adresu, a na kraju parametre koji se prosljeđuju funkciji. To znači da napadač može kontrolirati parametre. Uz osvježavanje kazaljke na kraju funkcije napadač je u mogućnosti prepisati proizvoljnu adresu sa vrijednosti `ptr`, što se događa prije izvršavanja kôda zaduženog za provjeru sigurnosnog kolačića. Na taj način napadač ima nekoliko mogućnosti napada u cilju zaobilaženja zaštite stoga.

3.3.1. Dvostruko prepisivanje sigurnosnog kolačića

Ukoliko se dogodi prepisivanje spremnika opisano ranije, napadač to može iskoristiti za prepisivanje referentne inačice sigurnosnog kolačića zapisane u `.data` dijelu memorije. Pod uvjetom da je referentni kolačić prepisan predvidljivom vrijednosti napadač može prepisati kopiju na stogu s istom vrijednošću, tako da sigurnosna provjera daje pozitivan rezultat, a pohranjena povratna adresa bude prihvaćena. Zaštita od ove vrste napada je jednostavna; dovoljno je 32 bita `.data` dijela memorije gdje je pohranjen referentni kolačić proglasiti dozvoljenim samo za čitanje. To je moguće jednostavno napraviti sljedećim pozivom:

```
VirtualProtect(&Cookie,0x04,PAGE_READONLY,&old);
```

Ovaj poziv potrebno je izvršiti čim se postavi sigurnosni kolačić. Mali nedostatak ove metode je što `VirtualProtect()` funkcija može zaštiti samo čitavu memorijsku stranicu, odnosno 4kB. Kolačić se mora postaviti na samu granicu stranice, dok se ostali podaci koje nije potrebno mijenjati mogu postaviti u preostali slobodni stranice.

3.3.2. Prepisivanje sigurnosnog rukovatelja

Ukoliko se sigurnosni kolačić na stogu ne podudara s referentnim kolačićem, vrši se provjera da li je definiran sigurnosni rukovatelj. Ukoliko je to slučaj, uz ranije opisanu funkciju, napadač može

prepisati adresu rukovatelja ukoliko je on definiran ili postaviti vlastiti rukovatelj ukoliko on nije bio definiran. Uz kazaljku koja pokazuje na napadački blok kôda ovakva mogućnost napada je izvediva. Unicode spremnik mora započeti s "\\" (`0x5C005C00`), pošto je potrebno da to bude UNC putanja koja će inicirati prepisivanje. Asemblirani okteti `0x5C005C00` predstavljaju sljedeći kôd:

```
POP ESP
ADD BYTE PTR[EAX+EAX+N]
```

gdje je N sljedeći oktet u spremniku. Pošto adresa `EAX+EAX+N` nije dozvoljena za pisanje, proces bi prekršio pravila za pisanje (engl. *access violation*) i napadač ne bi došao u priliku da dođe u kontrolu nad procesom.
Zaštitu od ovakvih napada moguće je implementirati na dva načina. Jedna od njih je izbjegavanje definiranja sigurnosnog rukovatelja i trenutno terminiranje procesa, dok druga mogućnost ponovno podrazumijeva postavljanje 32 bita `.data` memorije gdje se nalazi kazaljka na rukovatelj dozvoljen samo za čitanje:

```
VirtualProtect(&sec_func,0x04,PAGE_READONLY,&old);
```

Na ovaj način postiže se zaštita od prepisivanja, no ponovno se cijela memorijska stranica proglašava dozvoljenom samo za čitanje.

3.3.3. Zamjena kazaljke sistemskog direktorija

Ukoliko napadač kontrolira spremnik na koji pokazuje kazaljka, moguće je prepisati kazaljku na Windows sistemski direktorij koja je pohranjena u `.data` dijelu `kernel32.dll` biblioteke. Ako se sigurnosni kolačići ne podudaraju, poziva se funkcija `UnhandledExceptionFilter()`. Ta funkcija poziva funkciju `GetSystemDirectoryW()` koja koristi spomenutu kazaljku da bi došla do sistemskog direktorija. Ta vrijednost se tada kopira u spremnik s kojim je povezana `faultrep.dll` biblioteka. Navedena biblioteka se učitava i poziva se `ReportFault()` funkcija. Zamjenom kazaljke koja pokazuje na Windows sistemski direktorij s kazaljkom na korisnički definirani spremnik napadač može u proces učitati vlastitu `faultrep.dll` biblioteku u kojoj se nalazi njegov proizvoljno definirani kôd.

3.3.4. Prepisivanje kazaljki na Ldr*() funkcije

Postoji više `Ldr*()` funkcija, kao što su npr. `LoadLibrary()` i `FreeLibrary()`, koje se pozivaju prilikom provjere da li je postavljena funkcijska kazaljka. One su povezane sa Shim mehanizmom. Pošto `UnhandledExceptionFilter()` funkcija poziva `LoadLibraryExW()` i `FreeLibrary()` funkcije, napadač ima priliku postaviti te funkcijske kazaljke tako da mu omoguće dobivanje kontrole nad procesom. Npr., `FreeLibrary()` funkcija poziva `LdrUnloadDll()` funkciju u `NTDLL.DLL` biblioteci. Programski kôd te funkcije izgleda ovako:

```
77F5337A mov eax, [77FC2410]
77F5337F cmp eax, edi
77F53381 jne 77F53134
..
77F53134 push esi
77F53135 call eax
```

Ovaj dio kôda posprema vrijednost na adresi `0x77FC2410` u `.data` dijelu `NTDLL.DLL` biblioteke u `EAX` registar, te ga zatim uspoređuje s `0` (`EDI` registar sadrži vrijednost `0`). Drugim riječima, funkcijska kazaljka je postavljana na `0x77FC2410`, `ESI` se potiskuje na stog i funkcija se poziva. Postavljanjem ove funkcije napadač može doći u kontrolu nad procesom.
Zaštitu za oba ovakva napada, zamjenu kazaljke sistemskog direktorija i prepisivanje kazaljki funkcija, je teško implementirati pošto postoji ogroman broj potencijalnih napadačkih vektora, a sve dijelove memorije jednostavno nije moguće zaštititi od pisanja. Najbolji pristup bi bio kreirati novu funkciju jezgre `NTTerminateProcessOnStackOverflow()` i omogućiti njeno trenutno pozivanje prilikom detekcije prepisivanja (nepodudarnost sigurnosnih kolačića).

4. Zaključak

Ovaj dokument pokazuje da postoji nekoliko načina zaobilaženja zaštite stoga koja je implementirana unutar Windows Server 2003 operacijskog sustava, a potencijalno je postoje i druge mogućnosti.
Uvijek mogu postojati mogućnosti koje napadaču daju priliku dolaska u kontrolu nad pojedinim procesom, a potencijalno i čitavim sustavom, bez obzira kakve mehanizme zaštite pruža sam operacijski sustav. Prevencija je uvijek pouzdanija metoda od naknadnog djelovanja, tako da je nemoguće pronaći adekvatnu zamjenu za pridržavanje osnovnih sigurnosnih pravila prilikom samog kôdiranja.

Dodatak A

Sljedeća dva ispisa mogu se iskoristiti za umetanje kôda u drugi proces i učitavanje DLL biblioteke te izvršavanje funkcije. Funkcija pretražuje adresni prostor u potrazi za procesom čiji okteti formiraju instrukcije koje je moguće iskoristiti za zaobilaženje zaštite stoga korištenjem strukturiranog rukovanja iznimkama.
Nakon prevođenja moguće ih je pokrenuti iz komandne linije, a rezultati će biti pohranjeni u tekstualnoj datoteci.

```
/* Inject.c - compile C:\>cl /TC inject.c */
#include <stdio.h>
#include <windows.h>
DWORD __stdcall RThread(pfnktstuff);
#define MAXINJECTSIZE 4096
typedef struct fnktstuff_t
{
unsigned int LoadLibraryAddress;
unsigned int GetProcAddressAddress;
unsigned char Library[260];
unsigned char Function[260];
} fnktstuff;
DWORD __stdcall RThread( fnktstuff *pfs)
{
__asm{
mov edi,dword ptr[esp+08h] // Get Pointer to pfs
push edi
lea esi, dword ptr [edi+8] // Get Pointer to Library
push esi // Push onto the stack
call dword ptr[edi] // call LoadLibrary
cmp eax,0
je end
pop edi
lea esi, dword ptr [edi+268]
push esi
push eax
call dword ptr[edi+4]
cmp eax,0
je finish
call eax
jmp finish
end:
pop edi
finish:
}
return 0;
}
int main(int argc, char *argv[])
{
```

```
HANDLE hProcess=NULL;
HANDLE hRemoteThread=NULL;
int ProcessID=0;
void *FunctionAddress=0;
void *data=NULL;
HMODULE k=NULL;
fnktstuff fs;
if(argc !=4)
return printf("C:\\>%s PID Library Function\n",argv[0]);
k = LoadLibrary("kernel32.dll");
fs.GetProcAddressAddress = (unsigned
int)GetProcAddress(k,"GetProcAddress");
fs.LoadLibraryAddress = (unsigned
int)GetProcAddress(k,"LoadLibraryA");
ZeroMemory(fs.Library,260);
ZeroMemory(fs.Function,260);
strncpy(fs.Library,argv[2],256);
strncpy(fs.Function,argv[3],256);
ProcessID = atoi(argv[1]);
if(ProcessID == 0)
return printf("ProcessID is 0\n");
hProcess = OpenProcess(PROCESS_ALL_ACCESS,FALSE,ProcessID);
if(hProcess == NULL)
return printf("OpenProcess() failed with error:
%d\n",GetLastError());
printf("Process %d opened.\n",ProcessID);
FunctionAddress = VirtualAllocEx(hProcess,0,MAXINJECTSIZE
,MEM_COMMIT,PAGE_EXECUTE_READWRITE);
if(Functi onAddress == NULL)
{
printf("VirtualAllocEx() failed with error: %d\n",GetLastError());
CloseHandle(hProcess);
return 0;
}
printf("RemoteAddress is %.8x\n",FunctionAddress);
data = (unsigned char *) VirtualAllocEx(hProcess,0,MAXINJECTSIZE,
MEM_COMMIT,PAGE_READWRITE);
if(data == NULL)
{
printf("VirtualAllocEx() failed with error: %d\n",GetLastError());
CloseHandle(hProcess);
return 0;
}
printf("DataAddress is %.8x\n",data);
WriteProcessMemory(hProcess, FunctionAddress, &RThread,
MAXINJECTSIZE, 0);
WriteProcessMemory(hProcess, data, &fs, sizeof(fnktstuff), 0 );
hRemoteThread = CreateRemoteThread(hProcess,NULL,0,
FunctionAddress,data,0,NULL);
if(hRemoteThread == NULL)
{
printf("CreateRemoteThread() failed with error:
%d\n",GetLastError());
CloseHandle(hProcess);
return 0;
}
CloseHandle(hRemoteThread);
CloseHandle(hProcess);
```

```
return 0;
}
The code of the DLL:
// Compile: C:\>cl /LD code.dll
#include <stdio.h>
#include <windows.h>
int __declspec (dllexport) code()
{
FILE *fd = NULL;
unsigned char *p = 0;
unsigned char *c = (char *)0x7fffffff;
fd = fopen("results.txt","w+");
if(!fd)
return 0;
while(p < c)
{
__try{
if ((p[0] > 0x57) && (p[0] < 0x5F) && (p[1] > 0x57) && (p[1] < 0x5F)
&& (p[2] > 0xC1) &&
(p[2] < 0xC4))
fprintf(fd,"pop, pop, ret found at\t\t\t%.8X\n", p );
else if (p[0] == 0xFF)
{
if(p[1] == 0x55)
{
if(p[2] == 0x30)
fprintf(fd,"call dword ptr[ebp+30] found at\t\t%.8X\n", p );
else if(p[2] == 0x24)
fprintf(fd,"call dword ptr[ebp+24] found at\t\t%.8X\n", p );
else if(p[2] == 0x0C)
fprintf(fd,"call dword ptr[ebp+0C] found at\t\t%.8X\n", p );
else if(p[2] == 0xFC)
fprintf(fd,"call dword ptr[ebp-04] found at\t\t%.8X\n", p );
else if(p[2] == 0xF4)
fprintf(fd,"call dword ptr[ebp-0C] found at\t\t%.8X\n", p );
else if(p[2] == 0xE8)
fprintf(fd,"call dword ptr[ebp-18] found at\t\t%.8X\n", p );
}
else if(p[1] == 0x65)
{
if(p[2] == 0x30)
fprintf(fd,"jmp dword ptr[ebp+30] found at\t\t%.8X\n", p );
else if(p[2] == 0x24)
fprintf(fd,"jmp dword ptr[ebp+24] found at\t\t%.8X\n", p );
else if(p[2] == 0x0C)
fprintf(fd,"jmp dword ptr[ebp+0C] found at\t\t%.8X\n", p );
else if(p[2] == 0xFC)
fprintf(fd,"jmp dword ptr[ebp-04] found at\t\t%.8X\n", p );
else if(p[2] == 0xF4)
fprintf(fd,"jmp dword ptr[ebp-0C] found at\t\t%.8X\n", p );
else if(p[2] == 0xE8)
fprintf(fd,"jmp dword ptr[ebp-18] found at\t\t%.8X\n", p );
}
else if(p[1] == 0x54)
{
if(p[2] == 0x24 && p[3] == 0x08)
fprintf(fd,"call dword ptr[esp+08] found at\t\t%.8X\n", p );
else if(p[2] == 0x24 && p[3] == 0x14)
```

```
fprintf(fd,"call dword ptr[esp+14] found at\t\t%.8X\n", p );
else if(p[2] == 0x24 && p[3] == 0x1C)
fprintf(fd,"call dword ptr[esp+1C] found at\t\t%.8X\n", p );
else if(p[2] == 0x24 && p[3] == 0x2C)
fprintf(fd,"call dword ptr[esp+2C] found at\t\t%.8X\n", p );
else if(p[2] == 0x24 && p[3] == 0x44)
fprintf(fd,"call dword ptr[esp+44] found at\t\t%.8X\n", p );
else if(p[2] == 0x24 && p[3] == 0x50)
fprintf(fd,"call dword ptr[esp+50] found at\t\t%.8X\n", p );
}
else if(p[1] == 0x64)
{
if(p[2] == 0x24 && p[3] == 0x08)
fprintf(fd,"jmp dword ptr[esp+08] found at\t\t%.8X\n", p );
else if(p[2] == 0x24 && p[3] == 0x14)
fprintf(fd,"jmp dword ptr[esp+14] found at\t\t%.8X\n", p );
else if(p[2] == 0x24 && p[3] == 0x1C)
fprintf(fd,"jmp dword ptr[esp+1C] found at\t\t%.8X\n", p );
else if(p[2] == 0x24 && p[3] == 0x2C)
fprintf(fd,"jmp dword ptr[esp+2C] found at\t\t%.8X\n", p );
else if(p[2] == 0x24 && p[3] == 0x44)
fprintf(fd,"jmp dword ptr[esp+44] found at\t\t%.8X\n", p );
else if(p[2] == 0x24 && p[3] == 0x50)
fprintf(fd,"jmp dword ptr[esp+50] found at\t\t%.8X\n", p );
}
}
}
__except(EXCEPTION_EXECUTE_HANDLER)
{
p += 0x00000100;
}
p++;
}
fclose(fd);
return 0;
}
```