Lec 9: Control-flow Hijack

CS492E: Introduction to Software Security

Sang Kil Cha
Compilation


GNU AS (Assembler)

0: push ebp 0101010101101110110
1: mov ebp, esp 0000010111111010100
3: sub esp, 0x18 0010101001010111010
...

0101010101101110110
0000010111111010100
0010101001010111010
Understanding Binary

Source Code → Intermediate Code → Assembly Code

Disassembly (objdump)

0: push ebp
1: mov ebp, esp
3: sub esp, 0x18
...

Binary Code

01010101011011101100000101111110101000101010011001111010...

000010111110101000101001010111010
001010100101011101001010111010

a.out
Disassembly is Difficult

• Indirect jumps
  - `jmp [eax]`
  - `call [eax]`

• Mixture of code and data
Example: Linear Sweep Disassembly

```
.intel_syntax noprefix
inc ebx
sub eax, ebx
call lbl
.ascii "hello world"
lbl:
pop eax
```

Try to assemble it and disassemble the resulting binary with objdump
Recursive Traversal Disassembly

• a.k.a. Recursive Descent Disassembly
• Follow control-flows starting from some entry points
Complex Constructs

- Shared basic blocks
- Overlapping instructions
- Inlined data
- Alignment data
- Tail call optimization
- Etc.
Software Bug

= an error in a program
Question

What kind of bugs are important for security?

For example, if you only have time for fixing one bug out of ten, which bug will you fix first?
Exploitable Bug

We often call *exploitable bugs* as vulnerabilities.

*Exploitation* is the act of taking advantage of a bug to cause *unintended behavior* of the target program.

Some vulnerabilities allow an attacker to run any *arbitrary code* on victim’s machines without their consent.
Control Flow Hijack Exploit

Attacker’s own code
e.g., install malicious software
Classic Exploitation
Nov. 2, 1988

The first computer worm (called Morris Worm) was born.

Robert Tappan Morris
Creator of the worm
Cornell graduate
Tenured professor at MIT now
Morris Worm

Exploited a buffer overflow vulnerability in fingerd

```c
int main(int argc, char* argv[]) {
    char line[512];
    /* omitted ... */
    gets(line); /* Buffer Overflow! */
    /* omitted ... */
}
```

This single line allowed the Morris Worm to infect 10% of the Internet computers in 1988
Historic Exploitation

```c
int main(int argc, char* argv[]) {
    char line[512];
    gets(line);
    printf(line);
    return 0;
}
```

$ gcc -m32 -mpreferred-stack-boundary=2 -O0 -fno-pic -no-pie -z execstack -o morris morris.c

Compiler Warning (ignore this for now): morris.c:(.text+0x2a): warning: the `gets' function is dangerous and should not be used.

`gets(char *s)`
Reads a line from STDIN into the buffer pointed to by s until a terminating newline or EOF, which it replaces with a NULL byte (`\0`)
Historic Exploitation

```
int main(int argc, char* argv[])
{
    char line[512];
    gets(line);
    return 0;
}
```

$ gcc -m32 -mpreferred-stack-boundary=2 -O0 -fno-pic -no-pie -z execstack -o morris morris.c

**Compiler Warning (ignore this for now):**
morris.c:(.text+0x2a): warning: the `gets' function is dangerous and should not be used.

```
080483fb <main>:
     80483fb:    55                      push ebp
     80483fc:    89 e5                  mov ebp,esp
     80483fe:    81 ec 00 02 00 00        sub esp,0x200
     8048404:    8d 85 00 fe ff ff            lea eax,[ebp-0x200]
     804840a:    50                      push eax
     804840b:    e8 c0 fe ff ff            call 80482d0 <gets@plt>
     8048410:    83 c4 04                  add esp,0x4
     8048413:    b8 00 00 00 00 00        mov eax,0x0
     8048418:    c9                      leave
     8048419:    c3                      ret
```
080483fb <main>:
  080483fb: push ebp
  080483fc: mov ebp,esp
  080483fe: sub esp,0x200
  08048404: lea eax,[ebp-0x200]
  0804840a: push eax
  0804840b: call 80482d0 <gets@plt>
  08048410: add esp,0x4
  08048413: mov eax,0x0
  08048418: leave
  08048419: ret

Execution Context

esp = 0xbfffff70c
ebp = 0x0
eip = 0x80483fb

Return addr. (libc)
Old EBP (= 0)
080483fb <main>:
080483fb: push ebp
080483fc: mov ebp,esp
080483fe: sub esp,0x200
08048404: lea eax,[ebp-0x200]
0804840a: push eax
0804840b: call 80482d0 <gets@plt>
08048410: add esp,0x4
08048413: mov eax,0x0
08048418: leave
08048419: ret

Execution Context

esp = 0xbfffff708
ebp = 0x0
eip = 0x80483fc
080483fb <main>:
  80483fb: push   ebp
  80483fc: mov   ebp,esp
  80483fe: sub    esp,0x200
  8048404: lea   eax,[ebp-0x200]
  804840a: push   eax
  804840b: call   80482d0 <gets@plt>
  8048410: add   esp,0x4
  8048413: mov   eax,0x0
  8048418: leave
  8048419: ret

Execution Context

esp = 0xbffff708
ebp = 0xbffff708
eip = 0x80483fe

Return addr. (libc)
Old EBP (= 0)

512 byte

0xbffff70c
080483fb <main>:
80483fb: push ebp
80483fc: mov ebp, esp
80483fe: sub esp, 0x200
8048404: lea eax, [ebp - 0x200]
804840a: push eax
804840b: call 80482d0 <gets@plt>
8048410: add esp, 0x4
8048413: mov eax, 0x0
8048418: leave
8048419: ret

Execution Context

esp = 0xbfffff508
ebp = 0xbfffff708
eip = 0x8048404
eax = 0xbfffff508

Return addr. (libc)
Old EBP (= 0)
0x80483fb <main>:
80483fb: push ebp
80483fc: mov ebp, esp
80483fe: sub esp, 0x200
8048404: lea eax, [ebp - 0x200]
804840a: push eax
804840b: call 80482d0 <gets@plt>
8048410: add esp, 0x4
8048413: mov eax, 0x0
8048418: leave
8048419: ret

Execution Context:
- esp = 0xbffff508
- ebp = 0xbffff708
- eip = 0x804840a
- eax = 0xbffff508

Return addr. (libc)
Old EBP (= 0)

0xbffff70c
0xbffff508
Copy user input from STDIN to the buffer at 0xbffff508

Execution Context

- esp = 0xbffff504
- ebp = 0xbffff708
- eip = 0x804840b
- eax = 0xbffff508
Assume user input is 520 consecutive ‘A’s.

Execution Context:
- esp = 0xbffff504
- ebp = 0xbffff708
- eip = 0x804840b
- eax = 0xbffff508
080483fb <main>:
80483fb: push ebp
80483fc: mov ebp,esp
80483fe: sub esp,0x200
8048404: lea eax,[ebp-0x200]
804840a: push eax
804840b: call 80482d0 <gets@plt>
8048410: add esp,0x4
8048413: mov eax,0x0
8048418: leave
8048419: ret

Execution Context

esp = 0xbffff504
ebp = 0xbffff708
eip = 0x8048410
080483fb <main>:
80483fb: push ebp
80483fc: mov ebp, esp
80483fe: sub esp, 0x200
8048404: lea eax, [ebp - 0x200]
804840a: push eax
804840b: call 80482d0 <gets@plt>
8048410: add esp, 0x4
8048413: mov eax, 0x0
8048418: leave
8048419: ret

Execution Context

esp = 0xbfffff508
ebp = 0xbfffff708
eip = 0x8048413
eax = 0x0
080483fb <main>:
080483fb:  push  ebp
080483fc:  mov  ebp,esp
080483fe: sub  esp,0x200
08048404: lea  eax,[ebp-0x200]
0804840a: push  eax
0804840b: call  80482d0 <gets@plt>
08048410: add  esp,0x4
08048413: mov  eax,0x0
08048418: leave
08048419: ret

mov esp, ebp
pop ebp

Execution Context

esp  =  0xbffff508
ebp  =  0xbffff708
eip  =  0x8048418
eax  =  0x0
080483fb <main>:
80483fb: push ebp
80483fc: mov ebp,esp
80483fe: sub esp,0x200
8048404: lea eax,[ebp-0x200]
804840a: push eax
804840b: call 80482d0 <gets@plt>
8048410: add esp,0x4
8048413: mov eax,0x0
8048418: leave
8048419: ret

0xbffff70c ➔

Execution Context

<table>
<thead>
<tr>
<th>esp</th>
<th>0xbffff70c</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebp</td>
<td>0x41414141</td>
</tr>
<tr>
<td>eip</td>
<td>0x8048419</td>
</tr>
<tr>
<td>eax</td>
<td>0x0</td>
</tr>
</tbody>
</table>

Return addr. (libc)

Control Hijacked!

pop eip
So Far ...

• We hijacked the control of the program (= We can jump to anywhere!)

• But, where do we jump to?

• We want to inject some code to run!
This is so-called the **return-to-stack** exploit.
Executing *Shellcode*

- Small piece of code that is used as the payload

- Shellcode can run any arbitrary logic
  - Download /etc/passwd
  - Install malicious software (malware)
  - ...

- But typically executing /bin/sh is enough
  - This is the most powerful attack: we can run arbitrary commands
  - You can also achieve this with relatively *small amount of code*
  - This is the reason why we call it as shellcode (code that typically runs shell)
Write an Infinite Loop Shellcode

ồ.front

loop:  
jmp loop
Final Exploitation

• Fill the buffer with our shellcode (31 bytes)
• The rest of the buffer (481 bytes = 512-31) can be filled with any characters
• The old ebp can be filled with any characters (4 bytes)
• The return address should point to the shellcode (0xbfffff508)
Caveat

We assume that we know the exact address of the buffer.

This is very difficult even without modern defenses such as ASLR.
Practice: Same Machine w/ or w/o GDB

GDB reference:
http://www.yolinux.com/TUTORIALS/GDB-Commands.html

To see how to enable Intel syntax in GDB:
Add the following to ~/.gdbinit file
(set disassembly-flavor intel)
Why Different?

The key problem is *Environment Variables*

• GDB puts extra environment variables
• Each machine has different environment variables
Making Exploit Robust

• NOP sled (= NOP slide)

- 0x90 represents an 1-byte instruction NO-OP (= xchg eax, eax)

• Store a payload in an environment variable
  - We can control the size of the buffer (= we can put a larger NOP sled)
  - Works only for local exploitation
Off-by-One Error
```
#include <stdio.h>
#include <string.h>
#define BUFSIZE  (512)

void printer(char* str)
{
    char buf[BUFSIZE];
    strcpy(buf, str);
    printf("%s\n", buf);
}

int main(int argc, char* argv[])
{
    if ( argc < 2 || strlen(argv[1]) > BUFSIZE ) return -1;
    printer(argv[1]);
    return 0;
}
```

**Subtle Error**

We can just overwrite 1 byte NULL beyond the size of the buffer (buf)

But, some off-by-one bugs are exploitable!
```c
#include <stdio.h>
#include <string.h>
#define BUFSIZE  (512)

void printer(char* str)
{
    char buf[BUFSIZE];
    strcpy(buf, str);
    printf("%s\n", buf);
}

int main(int argc, char* argv[])
{
    if ( argc < 2 || strlen(argv[1]) > BUFSIZE ) return -1;
    printer(argv[1]);
    return 0;
}
```

Exercise: Can you draw the stack diagram?
```c
#include <stdio.h>
#include <string.h>
#define BUFSIZE (512)

void printer(char* str)
{
    char buf[BUFSIZE];
    strcpy(buf, str);
    printf("%s\n", buf);
}

int main(int argc, char* argv[])
{
    if ( argc < 2 || strlen(argv[1]) > BUFSIZE ) return -1;
    printer(argv[1]);
    return 0;
}
```
Lots of Defenses and Attacks Since 1988

- Data execution prevention
- Code reuse attacks
- Canary
- Address space layout randomization
- and many more …
Recommended Readings

• Smashing the Stack for Fun and Profit, Phrack 1996 by Aleph One
  http://phrack.org/issues/49/14.html

• x86 Calling Conventions
  https://en.wikipedia.org/wiki/X86_calling_conventions
Summary

• Only some bugs are exploitable.
• Some exploits allow an attacker to hijack the control flow of the target program and to run any arbitrary code.
• Return-to-stack exploit puts a shellcode in to a stack buffer and jumps to it by overwriting the return address.
• We can make return-to-stack exploit robust by using NOP sleds.
Questions?