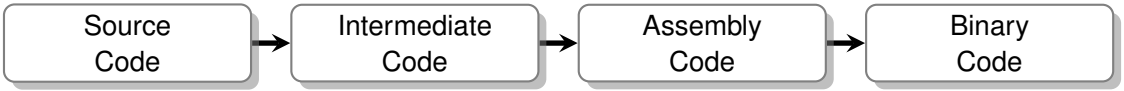
Lec 5: Control Flow Hijack

IS561: Binary Code Analysis and Secure Software Systems

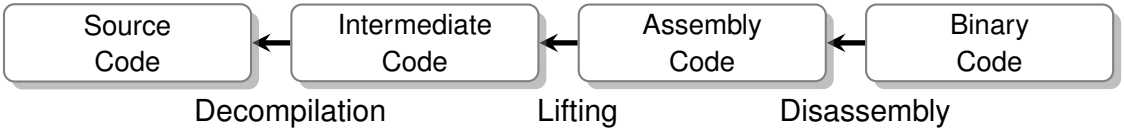
Sang Kil Cha

Disassembly Overview

Compilation Process



Reverse Engineering



Control Flow Hijack

Software Bug

Software bug is an ***error*** in a program.

Software Bug

Software bug is an ***error*** in a program.

Q: What kind of bugs are important for security? If you only have time for fixing one bug out of hundred, which bug will you fix first?

Exploitable Bugs

We often call an ***exploitable bug*** as a vulnerability.

Exploitable Bugs

We often call an ***exploitable bug*** as a vulnerability.

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

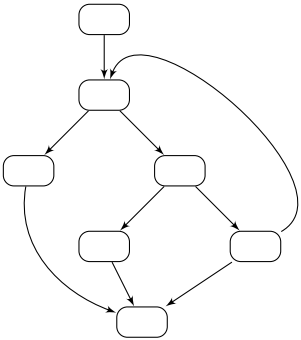
Exploitable Bugs

We often call an ***exploitable bug*** as a vulnerability.

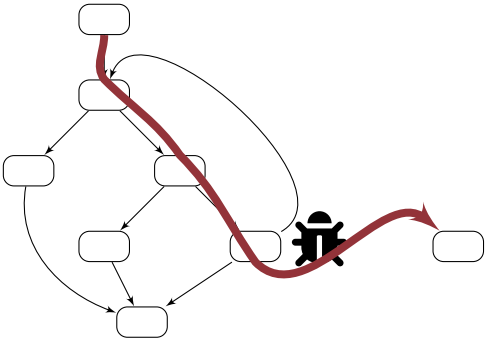
Exploitation is an act of taking advantage of a bug to cause ***unintended behavior*** of the target program.

Some vulnerabilities allow an attacker to run any ***arbitrary code*** on victim's machines without their consent.

Control Flow Hijack Exploit



Control Flow Hijack Exploit



Attacker's own code/logic

The Classic Exploitation

In 1988, the first computer worm (named 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[1] in `fingerd`.

Simplified `fingerd` vulnerability.

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

[1] This simple vulnerability affected 10% of the internet computers in 1988.

Replicating Historic Exploitation

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

Compile this program with:

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

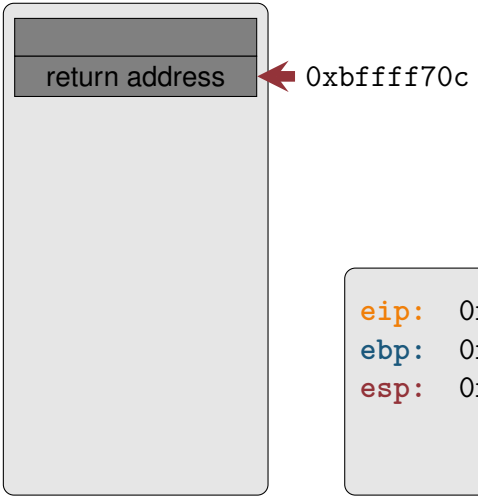
Compiler Warning: morris.c:(.text+0x2a): warning: the ‘gets’ function is dangerous and should not be used.

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’).

★ Type “`man gets`” to see the manual. What does it say?

Analyzing the Vulnerability



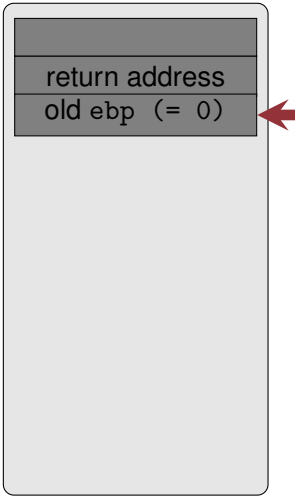
Virtual Memory

```
eip:  0x80483fb
ebp:  0x0
esp:  0xbffff70c
```

Execution Context

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

Analyzing the Vulnerability



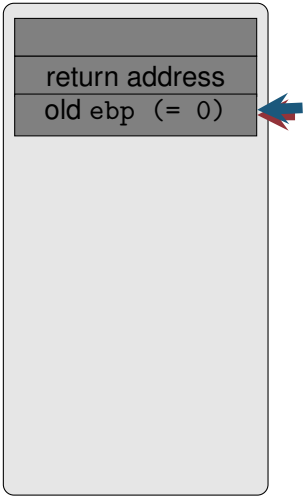
Virtual Memory

```
eip:  0x80483fc
ebp:  0x0
esp:  0xbffff708
```

Execution Context

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

Analyzing the Vulnerability



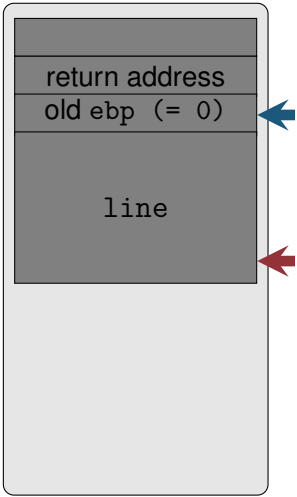
Virtual Memory

```
eip:  0x80483fe
ebp:  0xbffff708
esp:  0xbffff708
```

Execution Context

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

Analyzing the Vulnerability



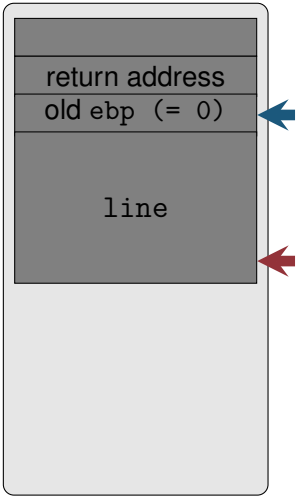
Virtual Memory

```
eip:  0x8048404
ebp:  0xbffff708
esp:  0xbffff508
```

Execution Context

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

Analyzing the Vulnerability



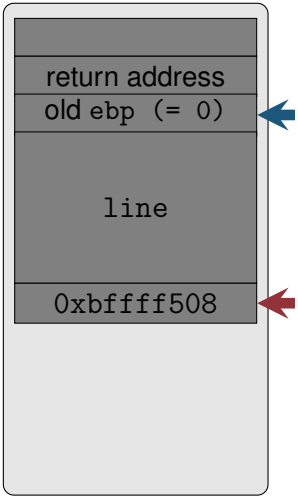
Virtual Memory

```
eip:  0x804840a
ebp:  0xbffff708
esp:  0xbffff508
eax:  0xbffff508
```

Execution Context

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

Analyzing the Vulnerability



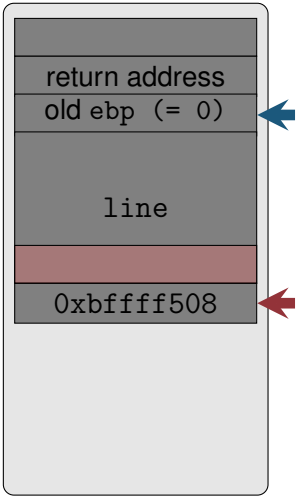
Virtual Memory

```
eip:  0x804840b
ebp:  0xbffff708
esp:  0xbffff504
eax:  0xbffff508
```

Execution Context

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

Analyzing the Vulnerability



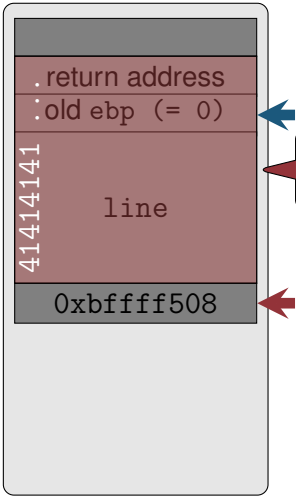
Virtual Memory

```
eip:  0x804840b
ebp:  0xbffff708
esp:  0xbffff504
eax:  0xbffff508
```

Execution Context

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

Analyzing the Vulnerability



Assume user input is 520 consecutive ‘A’s

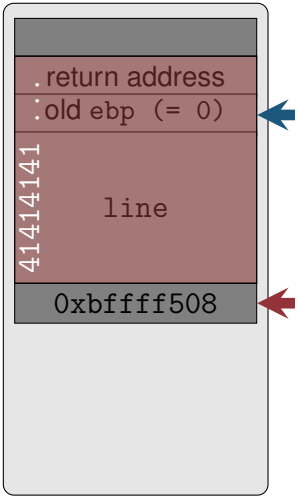
```
80483fb <main>:
80483fb: push ebp
80483fc: mov ebp,esp
80483fe: sub esp,0x200
8048404: lea eax,[ebp-0x200]
804840a: push eax
804840b: call 80482d0 ; gets
8048410: add esp,0x4
8048413: mov eax,0x0
8048418: leave
8048419: ret
```

```
eip:  0x804840b
ebp:  0xbffff708
esp:  0xbffff504
eax:  0xbffff508
```

Virtual Memory

Execution Context

Analyzing the Vulnerability



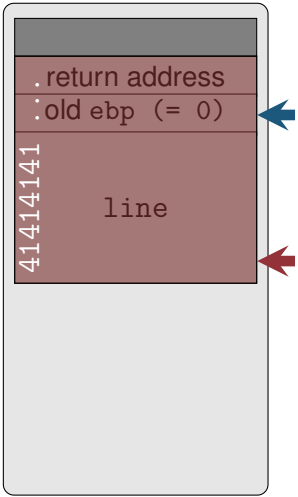
Virtual Memory

```
eip:  0x8048410
ebp:  0xbffff708
esp:  0xbffff504
```

Execution Context

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

Analyzing the Vulnerability



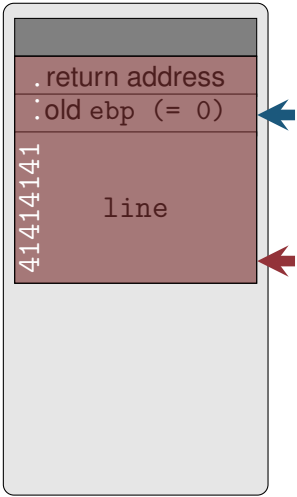
Virtual Memory

```
eip:  0x8048413
ebp:  0xbffff708
esp:  0xbffff508
```

Execution Context

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

Analyzing the Vulnerability



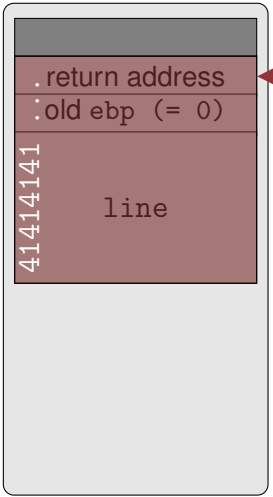
Virtual Memory

```
eip:  0x8048418
ebp:  0xbffff708
esp:  0xbffff508
eax:  0x0
```

Execution Context

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

Analyzing the Vulnerability



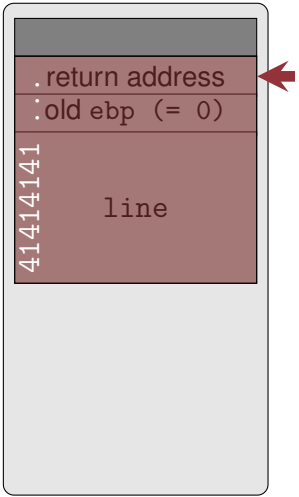
Virtual Memory

```
eip:  0x8048419
ebp:  0x41414141
esp:  0xbffff708
eax:  0x0
```

Execution Context

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

Analyzing the Vulnerability



Virtual Memory

```
eip:  0x41414141
ebp:  0x0
esp:  0xbffff708
eax:  0x0
```

Execution Context

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

```
41414141:  ???
```

So Far ...

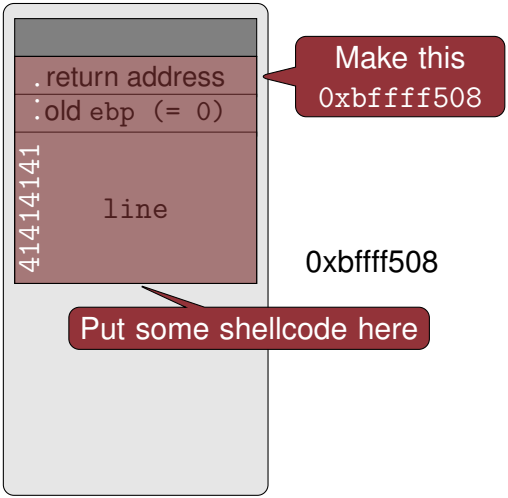
- We hijacked the control flow of the program, i.e., we can jump to anywhere!
- But, where do we jump to?

So Far ...

- We hijacked the control flow of the program, i.e., we can jump to anywhere!
- But, where do we jump to?

Injecting code to run

Return-to-Stack Exploit



Virtual Memory

Recall Shellcode Requirement

Shellcode should run regardless of the address it is loaded. In other words, it should be ***position independent***.

What to Run?

- Shellcode can run any arbitrary logic
  - Download `/etc/passwd`.
  - Install malicious software (malware).
  - etc.
- But executing `/bin/sh` is mostly enough.
  - This is the most powerful attack: we can run arbitrary commands.
  - You can achieve this with relatively ***small amount of code***.
  - This is the reason why we call it as ***shellcode***.

Simplistic Shellcode: Infinite Loop Shellcode

Writing a shellcode for spawning `/bin/sh` is your homework. Let's use a simpler shellcode here.

```
.intel_syntax noprefix
loop:
        jmp loop
```

Final Exploitation

- Put our shellcode (2 bytes) at the beginning of the buffer.
- Fill the rest of the buffer (510 bytes) with any character (e.g., ‘A’s).
- Overwrite the old `ebp` on the stack with any character.
- Overwrite the return address to point to the shellcode (`0xbffff508`)[2].

[2]The buffer address should differ from machine to machine. Thus, it is necessary to obtain the right address from a debugger (e.g., GDB).

Caveat

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

But, this is not always possible!

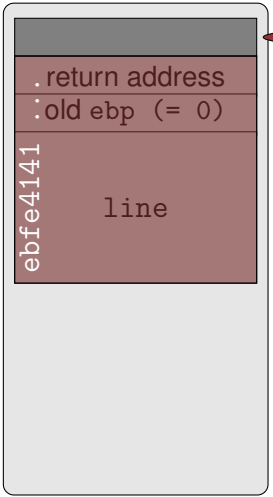
Using GDB

- GDB reference: `http://www.yolinux.com/TUTORIALS/GDB-Commands.html`.
- It is recommended to always turn on the Intel syntax by modifying "`$HOME/.gdbinit`" file.

Exploit w/ or w/o GDB

- The buffer address identified through GDB is not the same as it without GDB.
- Thus, our exploit doesn't work outside GDB!

Why Different?



This space is allocated for storing
environment variables

0xbffff508?

- GDB puts extra environment variables.
- Different machines have different environment variables.

Making Exploit Robust: NOP Sled

NOP sled (a.k.a. NOP slide) is used to make the exploit robust against different buffer addresses.

- One-byte NO-OP (NOP) instruction is equivalent to `xchg eax, eax`.
- 0x90 represents the NOP instruction.

Making Exploit Robust: Environment Variable

This is only for local (non-remote) exploitation.

- We can use an environment variable to store a large payload with a large NOP sled.
- This is useful when the input buffer size is limited.

Off-by-One Error

Subtle Error

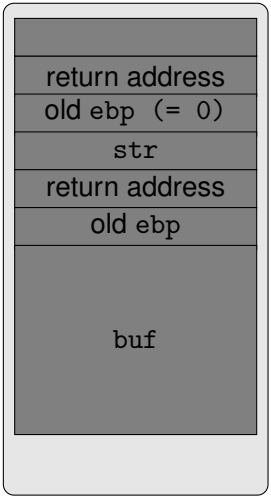
```c
#include <stdio.h>
#include <string.h>
#define BUFSIZE  (512)
void printer(char* str)
{
  char buf[BUFSIZE];
  strcpy(buf, str);
  printf("%s\n", buf);
}
int main(int argc, char* argv[])
{
  if ( argc < 2 || strlen(argv[1]) > BUFSIZE ) return -1;
  printer(argv[1]);
  return 0;
}
```

Subtle 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;
}
```

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

Off-by-one Bugs Can be Exploitable



Virtual Memory

`leave` = `mov esp, ebp; pop ebp`

Recommended Reading

Smashing the Stack for Fun and Profit, Phrack 1996, by Alphe One

- `http://phrack.org/issues/49/14.html`

Summary

- Only some bugs are exploitable.
- Some exploits allow an attacker to hijack the control flow of the program and to run any arbitrary code.
- Return-to-stack exploit puts a shellcode on the stack and jumps to it by overwriting the return address.
- We can make return-to-stack exploits robust by using NOP sleds.
- Off-by-one errors can sometimes be exploitable.

Question?

Exercise

- Write your own off-by-one error exploit for the example program.