CSE 127: Computer Security

Stack Buffer Overflows

Deian Stefan
Some slides adopted from Kirill Levchenko and Stefan Savage
When is a program secure?

• When it does exactly what it should?
  ➤ Not more
  ➤ Not less
When is a program secure?

- When it does exactly what it should?
  - Not more
  - Not less
- But how do we know what a program is supposed to do?
  - Somebody tells us? (Do we trust them?)
  - We write the code ourselves? (What fraction of the software you use have you written?)
When is a program secure?

- Try 2: When it doesn’t do bad things
- Easier to specify a list of “bad” things:
  - Delete or corrupt important files
  - Crash my system
  - Send my password over the Internet
  - Send threatening email to the professor
When is a program secure?

But ... what if most of the time the program doesn’t do bad things, but occasionally it does? Or could? Is it secure?
Weird machines

- Complex systems almost always contain unintended functionality
  - "Weird machines"
- An exploit is a mechanism by which an attacker triggers unintended functionality in the system
  - Programming of the weird machine
Weird machines

• Security requires understanding not just the intended but also the unintended functionality present in the implementation
  ➢ Developers’ blind spot
  ➢ Attackers’ strength
What is a software vulnerability?
What is a software vulnerability?

- A bug in a program that allows an unprivileged user capabilities that should be denied to them
What is a software vulnerability?

• A bug in a program that allows an unprivileged user capabilities that should be denied to them

• There are a lot of types of vulns, but among the most classic and important are vulnerabilities that violate “control flow integrity”

➤ Why? Lets attacker run code on your computer!
What is a software vulnerability?

• A bug in a program that allows an unprivileged user capabilities that should be denied to them

• There are a lot of types of vulns, but among the most classic and important are vulnerabilities that violate “control flow integrity”

➤ Why? Let’s attacker run code on your computer!

• Typically these involve violating assumptions of the programming language or its run-time
Starting exploits

• Dive into low level details of how exploits work
  ➤ How can a remote attacker get your machine to execute their code?

• Threat model
  ➤ Victim code is handling input that comes from across a security boundary
    ➤ What are some examples of this?
  ➤ Want to protect integrity of execution and confidentiality of data from being compromised by malicious and highly skilled users of our system.
Today: (stack) buffer overflows

Lecture objectives:

➤ Understand how buffer overflow vulnerabilities can be exploited
➤ Identify buffer overflow vulnerabilities in code and assess their impact
➤ Avoid introducing buffer overflow vulnerabilities during implementation
➤ Correctly fix buffer overflow vulnerabilities
Buffer overflows

- **Defn:** an anomaly that occurs when a program writes data beyond the boundary of a buffer.

- Archetypal software vulnerability
  - Ubiquitous in system software (C/C++)
    - OSes, web servers, web browsers, etc.
  - If your program crashes with memory faults, you probably have a buffer overflow vulnerability.
Why are they interesting?

• A basic core concept that enables a broad range of possible attacks
  ➤ Sometimes a single byte is all the attacker needs

• Ongoing arms race between defenders and attackers
  ➤ Co-evolution of defenses and exploitation techniques
How are they introduced?
How are they introduced?

- No automatic bounds checking in C/C++
How are they introduced?

- No automatic bounds checking in C/C++
- The problem is made more acute by the fact many C stdlib functions make it easy to go past bounds
- String manipulation functions like `gets()`, `strcpy()`, and `strcat()` all write to the destination buffer until they encounter a terminating ‘\0’ byte in the input
How are they introduced?

- No automatic bounds checking in C/C++
- The problem is made more acute by the fact many C stdlib functions make it easy to go past bounds
- String manipulation functions like `gets()`, `strcpy()`, and `strcat()` all write to the destination buffer until they encounter a terminating ‘\0’ byte in the input
  - Whoever is providing the input *(often from the other side of a security boundary)* controls how much gets written
Example 1: spot the vuln!

```c
main(argc, argv)
  char *argv[];
{
  register char *sp;
  char line[512];
  struct sockaddr_in sin;
  int i, p[2], pid, status;
  FILE *fp;
  char *av[4];

  i = sizeof (sin);
  if (getpeername(0, &sin, &i) < 0)
    fatal(argv[0], "getpeername");
  line[0] = '\0';
  gets(line);
  //...
  return(0);
}
```

http://minnie.tuhs.org/cgi-bin/utree.pl?file=4.3BSD/usr/src/etc/fingerd.c
Example 1: spot the vuln!

- What does `gets()` do?

```c
main(argc, argv)
char *argv[];
{
    register char *sp;
    char line[512];
    struct sockaddr_in sin;
    int i, p[2], pid, status;
    FILE *fp;
    char *av[4];

    i = sizeof(sin);
    if (getpeername(0, &sin, &i) < 0)
        fatal(argv[0], "getpeername");
    line[0] = '\0';
    gets(line);
    //...
    return(0);
}
```

http://minnie.tuhs.org/cgi-bin/utree.pl?file=4.3BSD/usr/src/etc/fingerd.c
Example 1: spot the vuln!

- What does `gets()` do?
  - How many characters does it read in?
  - Who decides how much input to provide?

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    register char *sp;
    char line[512];
    struct sockaddr_in sin;
    int i, p[2], pid, status;
    FILE *fp;
    char *av[4];

    i = sizeof (sin);
    if (getpeername(0, &sin, &i) < 0)
        fatal(argv[0], "getpeername");
    line[0] = '\0';
    gets(line);
    //...
    return(0);
}
```

http://minnie.tuhs.org/cgi-bin/utree.pl?file=4.3BSD/usr/src/etc/fingerd.c
Example 1: spot the vuln!

• What does `gets()` do?
  ➤ How many characters does it read in?
  ➤ Who decides how much input to provide?

• How large is `line[]`?

```c
main(argc, argv)
char *argv[];
{
  register char *sp;
  char line[512];
  struct sockaddr_in sin;
  int i, p[2], pid, status;
  FILE *fp;
  char *av[4];

  i = sizeof (sin);
  if (getpeername(0, &sin, &i) < 0)
    fatal(argv[0], "getpeername");
  line[0] = '\\0';
  gets(line);
  //...
  return(0);
}
```
Example 1: spot the vuln!

- What does `gets()` do?
  - How many characters does it read in?
  - Who decides how much input to provide?
- How large is `line[]`?
  - Implicit assumption about input length

```c
main(argc, argv)
char *argv[];
{
    register char *sp;
    char line[512];
    struct sockaddr_in sin;
    int i, *p[2], pid, status;
    FILE *fp;
    char *av[4];

    i = sizeof (sin);
    if (getpeername(0, &sin, &i) < 0)
        fatal(argv[0], "getpeername");
    line[0] = '\0';
    gets(line);
    //...
    return(0);
}
```

http://minnie.tuhs.org/cgi-bin/utree.pl?file=4.3BSD/usr/src/etc/fingerd.c
Example 1: spot the vuln!

- What does gets() do?
  - How many characters does it read in?
  - Who decides how much input to provide?

- How large is line[]?
  - Implicit assumption about input length

- What happens if, say 536, characters are provided as input?

```c
main(argc, argv)
char *argv[];
{
    register char *sp;
    char line[512];
    struct sockaddr_in sin;
    int i, p[2], pid, status;
    FILE *fp;
    char *av[4];

    i = sizeof (sin);
    if (getpeername(0, &sin, &i) < 0)
        fatal(argv[0], "getpeername");
    line[0] = '\'0';
    gets(line);
    //...
    return(0);
}
```
Morris worm

- This fingered vulnerability was one of several exploited by the Morris Worm in 1988
  - Created by Robert Morris graduate student at Cornell
- One of the first Internet worms
  - Devastating effect on the Internet at the time
  - Took over hundreds of computers and shut down large chunks of the Internet
- Aside: First use of the US CFAA

https://en.wikipedia.org/wiki/Morris_worm
But it’s 2019... still a problem?

**Issue 930035: Security: Stack out-of-bounds writes in WebmMuxer::AddAudioTrack**
Reported by mifbr...@stanford.edu on Thu, Feb 7, 2019, 9:30 PM PST

**Vulnerability Details**
Security: Stack out-of-bounds writes in WebmMuxer::AddAudioTrack
I'm not certain that its triggerable, but it looks as if it should be fixed

**Reproduction Case**

1. Stack allocated opus header in AddAudioTrack:
   ```
   uint8_t opus_header[OPUS_EXTRADATA_SIZE]; // and OPUS_EXTRADATA_SIZE is 19
   (see: https://cs.chromium.org/chromium/src/media/muxers/webm_muxer.cc?type=cs&q=WebmMuxer::AddAudioTrack&g=0&l=303)
   ```

2. Function calls WriteOpusHeader(params, opus_header);
   (function https://cs.chromium.org/chromium/src/media/muxers/webm_muxer.cc?l=20&gsn=WriteOpusHeader)

3. WriteOpusHeader does:
   ```
   header[OPUS_EXTRADATA_NUM_COUPLED_OFFSET] = 0;
   ```
   where
   ```
   OPUS_EXTRADATA_NUM_STREAMS_OFFSET = OPUS_EXTRADATA_SIZE,
   OPUS_EXTRADATA_NUM_COUPLED_OFFSET = OPUS_EXTRADATA_NUM_STREAMS_OFFSET + 1,
   OPUS_EXTRADATA_STREAM_MAP_OFFSET = OPUS_EXTRADATA_NUM_STREAMS_OFFSET + 2
   (allocated size was OPUS_EXTRADATA_SIZE)
   ```

4. It continues to write out of bounds:
   ```
   for (int i = 0; i < params.channels(); ++i) {
     header[OPUS_EXTRADATA_STREAM_MAP_OFFSET + i] =
     kOpusVorbisChannelMap[params.channels() + 1][i];
   }
   ```
OK but…

• Why does overflowing a buffer let you take over a machine?

• That seems crazy no?
Changing perspectives

• Your program manipulates data
• Data manipulates your program
What we need to know

• How C arrays work
• How memory is laid out
• How function calls work
• How to turn an array overflow into an exploit
How does an array work?
How does an array work?

• What's the abstraction?

```
a[i]
a[3]
a[0]
a[-i]
```
How does an array work?

• What's the abstraction?

• What’s the reality?
  ➤ What happens if you try to write past the of an array in C/C++?

```c
a[i]
a[3]
a[0]
a[-i]
```
How does an array work?

• What's the abstraction?

• What’s the reality?
  ➤ What happens if you try to write past the of an array in C/C++?
  ➤ What does the language spec say?
How does an array work?

- What's the abstraction?
- What's the reality?
  - What happens if you try to write past the end of an array in C/C++?
  - What does the language spec say?
  - What happens in most implementations?
Linux process memory layout

- Stack
- Heap
- Data segment
  - .data, .bss
- Text segment
  - Executable code
The Stack

- Stack divided into frames
  - Frame stores locals and args to called functions
- **Stack pointer** points to top of stack
  - x86: Stack grows down (from high to low addresses)
  - x86: Stored in %esp register
- **Frame pointer** points to caller’s stack frame
  - Also called base pointer
  - x86: Stored in %ebp register
Stack frame

- to previous frame pointer
- arguments
- return address
- stack frame pointer
- local variables

Stack growth

To instruction that follows the call of this function
Example 0

```c
int foobar(int a, int b, int c)
{
    int xx = a + 2;
    int yy = b + 3;
    int zz = c + 4;
    int sum = xx + yy + zz;

    return xx * yy * zz + sum;
}

int main()
{
    return foobar(77, 88, 99);
}
```

https://godbolt.org/z/3iFhjy
Compiled to x86

```c
int foobar(int a, int b, int c)
{
    int xx = a + 2;
    int yy = b + 3;
    int zz = c + 4;
    int sum = xx + yy + zz;

    return xx * yy * zz + sum;
}

int main()
{
    return foobar(77, 88, 99);
}
```

```assembly
foobar(int, int, int):
pushl %ebp
movl %esp, %ebp
subl $16, %esp
movl 8(%ebp), %eax
addl $2, %eax
movl %eax, -4(%ebp)
movl 12(%ebp), %eax
addl $3, %eax
movl %eax, -8(%ebp)
movl 16(%ebp), %eax
addl $4, %eax
movl %eax, -12(%ebp)
movl -4(%ebp), %edx
movl -8(%ebp), %eax
addl %eax, %edx
movl -12(%ebp), %eax
addl %edx, %eax
movl %eax, -16(%ebp)
movl -4(%ebp), %eax
imull -8(%ebp), %eax
imull -12(%ebp), %eax
movl %eax, %edx
movl -16(%ebp), %eax
addl %edx, %eax
leave
ret

main:
pushl %ebp
movl %esp, %ebp
pushl $99
pushl $88
pushl $77
call foobar(int, int, int)
addl $12, %esp
nop
leave
ret
```
```plaintext
foobar(int, int, int):
  pushl %ebp
  movl %esp, %ebp
  subl $16, %esp
  movl 8(%ebp), %eax
  addl $2, %eax
  movl %eax, -4(%ebp)
  movl 12(%ebp), %eax
  addl $3, %eax
  movl %eax, -8(%ebp)
  movl 16(%ebp), %eax
  addl $4, %eax
  movl %eax, -12(%ebp)
  movl -4(%ebp), %edx
  movl -8(%ebp), %eax
  addl %eax, %edx
  movl -12(%ebp), %eax
  addl %edx, %eax
  movl %eax, -16(%ebp)
  movl -4(%ebp), %eax
  imull -8(%ebp), %eax
  imull -12(%ebp), %eax
  movl %eax, %edx
  movl -16(%ebp), %eax
  addl %edx, %eax
  leave
  ret

main:
  pushl %ebp
  movl %esp, %ebp
  pushl $99
  pushl $88
  pushl $77
  call foobar(int, int, int)
  addl $12, %esp
  nop
  leave
  ret
```
foo_bar(int, int, int):
        pushl %ebp
        movl %esp, %ebp
        subl $16, %esp
 movl @(%ebp), %eax
 addl $2, %eax
 movl %eax, -4(%ebp)
movl 12(%ebp), %eax
 addl $3, %eax
 movl %eax, -8(%ebp)
movl 16(%ebp), %eax
 addl $4, %eax
 movl %eax, -12(%ebp)
movl -4(%ebp), %edx
 movl -8(%ebp), %eax
 imull -8(%ebp), %eax
 imull -12(%ebp), %eax
 movl %eax, %edx
 movl -16(%ebp), %eax
 addl %edx, %eax
 leave
 ret
main:
        pushl %ebp
        movl %esp, %ebp
 pushl $99
 pushl $88
 pushl $77
 call foo_bar(int, int, int)
 addl $12, %esp
 nop
 leave
 ret
fooobar(int, int, int):
    pushl %ebp
    movl %esp, %ebp
    subl $16, %esp
    movl 8(%ebp), %eax
    addl $2, %eax
    movl %eax, -4(%ebp)
    movl 12(%ebp), %eax
    addl $3, %eax
    movl %eax, -8(%ebp)
    movl 16(%ebp), %eax
    addl $4, %eax
    movl %eax, -12(%ebp)
    movl -4(%ebp), %edx
    movl -8(%ebp), %eax
    addl %eax, %edx
    movl -12(%ebp), %eax
    addl %edx, %eax
    movl %eax, -16(%ebp)
    movl -4(%ebp), %eax
    imull -8(%ebp), %eax
    imull -12(%ebp), %eax
    movl %eax, %edx
    movl -16(%ebp), %eax
    addl %edx, %eax
    leave
    ret

main:
    pushl %ebp
    movl %esp, %ebp
    pushl $99
    pushl $88
    pushl $77
    call fooobar(int, int, int)
    addl $12, %esp
    nop
    leave
    ret
fooBar(int, int, int):
pushl %ebp
movl %esp, %ebp
subl $16, %esp
movl 8(%ebp), %eax
addl $2, %eax
movl %eax, -4(%ebp)
movl 12(%ebp), %eax
addl $3, %eax
movl %eax, -8(%ebp)
movl 16(%ebp), %eax
addl $4, %eax
movl %eax, -12(%ebp)
movl -4(%ebp), %edx
movl -8(%ebp), %eax
addl %eax, %edx
movl -12(%ebp), %eax
addl %edx, %eax
movl %eax, -16(%ebp)
imul -8(%ebp), %eax
imul -12(%ebp), %eax
movl %eax, %edx
movl -16(%ebp), %eax
addl %edx, %eax
leave
ret

main:
pushl %ebp
movl %esp, %ebp
pushl $99
pushl $88
pushl $77
call fooBar(int, int, int)
addl $12, %esp
nop
leave
ret
```c
foobar(int, int, int):
    pushl %ebp
    movl %esp, %ebp
    subl $16, %esp
    movl 8(%ebp), %eax
    addl $2, %eax
    movl %eax, -4(%ebp)
    movl 12(%ebp), %eax
    addl $3, %eax
    movl %eax, -8(%ebp)
    movl 16(%ebp), %eax
    addl $4, %eax
    movl %eax, -12(%ebp)
    movl -4(%ebp), %edx
    movl -8(%ebp), %eax
    addl %eax, %edx
    movl -12(%ebp), %eax
    addl %edx, %eax
    movl %eax, -16(%ebp)
    movl -4(%ebp), %eax
    imull -8(%ebp), %eax
    imull -12(%ebp), %eax
    movl %eax, %edx
    movl -16(%ebp), %eax
    addl %edx, %eax
    leave
    ret

main:
    pushl %ebp
    movl %esp, %ebp
    pushl $99
    pushl $88
    pushl $77
    call foobar(int, int, int)
    addl $12, %esp
    nop
    leave
    ret
```

%ebp → $99
$88
$77
%esp → 0xfffffd0d8
%eip = 0x8049ba7
```c
# foobar(int, int, int):
pushl %ebp
movl %esp, %ebp
subl $16, %esp
movl 8(%ebp), %eax
addl $2, %eax
movl %eax, -4(%ebp)
movl 12(%ebp), %eax
addl $3, %eax
movl %eax, -8(%ebp)
movl 16(%ebp), %eax
addl $4, %eax
movl %eax, -12(%ebp)
movl -4(%ebp), %edx
addl %eax, %edx
movl -12(%ebp), %eax
addl %edx, %eax
movl %eax, -16(%ebp)
imull -8(%ebp), %eax
imull -12(%ebp), %eax
movl %eax, %edx
movl -16(%ebp), %eax
addl %edx, %eax
leave
ret

# main:
pushl %ebp
movl %esp, %ebp
pushl $99
pushl $88
pushl $77
call foobar(int, int, int)
addl $12, %esp
nop
leave
ret
```
foo(bar(int, int, int)):
    pushl %ebp
    movl %esp, %ebp
    subl $16, %esp
    movl 8(%ebp), %eax
    addl $2, %eax
    movl %eax, -4(%ebp)
    movl 12(%ebp), %eax
    addl $3, %eax
    movl %eax, -8(%ebp)
    movl 16(%ebp), %eax
    addl $4, %eax
    movl %eax, -12(%ebp)
    movl -4(%ebp), %edx
    movl -8(%ebp), %eax
    addl %edx, %edx
    movl -12(%ebp), %eax
    addl %edx, %eax
    movl %eax, -16(%ebp)
    movl -4(%ebp), %eax
    imull -8(%ebp), %eax
    imull -12(%ebp), %eax
    movl %eax, %edx
    movl -16(%ebp), %eax
    addl %edx, %eax
    leave
    ret

main:
    pushl %ebp
    movl %esp, %ebp
    pushl $99
    pushl $88
    pushl $77
    call foo(bar(int, int, int))
    addl $12, %esp
    nop
    leave
    ret
fooBar(int, int, int):
  pushl %ebp
  movl %esp, %ebp
  subl $16, %esp
  movl 8(%ebp), %eax
  addl $2, %eax
  movl %eax, -4(%ebp)
  movl 12(%ebp), %eax
  addl $3, %eax
  movl %eax, -8(%ebp)
  movl 16(%ebp), %eax
  addl $4, %eax
  movl %eax, -12(%ebp)
  movl -4(%ebp), %edx
  addl %eax, %edx
  movl -12(%ebp), %eax
  addl %edx, %eax
  movl %eax, -16(%ebp)
  movl -4(%ebp), %eax
  imull -8(%ebp), %eax
  imull -12(%ebp), %eax
  movl %eax, %edx
  movl -16(%ebp), %eax
  addl %edx, %eax
leave
ret

main:
  pushl %ebp
  movl %esp, %ebp
  pushl $99
  pushl $88
  pushl $77
  call fooBar(int, int, int)
  addl $12, %esp
  nop
leave
ret
```c
foobar(int, int, int):
    pushl  %ebp
    movl  %esp, %ebp
    subl  $16, %esp
    movl  0(%ebp), %eax
    addl  $2, %eax
    movl  %eax, -4(%ebp)
    movl  12(%ebp), %eax
    addl  $3, %eax
    movl  %eax, -8(%ebp)
    movl  16(%ebp), %eax
    addl  $4, %eax
    movl  %eax, -12(%ebp)
    movl  -4(%ebp), %edx
    movl  -8(%ebp), %eax
    movl  -12(%ebp), %eax
    addl  %edx, %eax
    movl  %eax, -16(%ebp)
    movl  -4(%ebp), %eax
    imull  -8(%ebp), %eax
    imull  -12(%ebp), %eax
    movl  %eax, %edx
    movl  -16(%ebp), %eax
    addl  %edx, %eax
    leave
    ret

main:
    pushl  %ebp
    movl  %esp, %ebp
    pushl  $99
    pushl  $88
    pushl  $77
    call  foobar(int, int, int)
    addl  $12, %esp
    nop
    leave
    ret
```
```c
foobar(int, int, int):
    push1 %ebp
    movl %esp, %ebp
    subl $16, %esp
    movl 8(%ebp), %eax
    addl $2, %eax
    movl %eax, -4(%ebp)
    movl 12(%ebp), %eax
    addl $3, %eax
    movl %eax, -8(%ebp)
    movl 16(%ebp), %eax
    addl $4, %eax
    movl %eax, -12(%ebp)
    movl -4(%ebp), %edx
    addl %eax, %edx
    movl -12(%ebp), %eax
    addl %edx, %eax
    movl %eax, -16(%ebp)
    movl -4(%ebp), %eax
    imull -8(%ebp), %eax
    imull -12(%ebp), %eax
    movl %eax, %edx
    movl -16(%ebp), %eax
    addl %edx, %eax
    leave
    ret

main:
    push1 %ebp
    movl %esp, %ebp
    pushl $99
    pushl $88
    pushl $77
    call foobar(int, int, int)
    addl $12, %esp
    nop
    leave
    ret
```
```c
foobar(int, int, int):
    pushl %ebp
    movl %esp, %ebp
    subl $16, %esp
    movl 8(%ebp), %eax
    addl $2, %eax
    movl %eax, -4(%ebp)
    movl 12(%ebp), %eax
    addl $3, %eax
    movl %eax, -8(%ebp)
    movl 16(%ebp), %eax
    addl $4, %eax
    movl %eax, -12(%ebp)
    movl -4(%ebp), %edx
    movl -8(%ebp), %eax
    addl %eax, %edx
    movl -12(%ebp), %eax
    addl %edx, %eax
    movl %eax, -16(%ebp)
    imull -8(%ebp), %eax
    imull -12(%ebp), %eax
    movl %eax, %edx
    movl -16(%ebp), %eax
    addl %edx, %eax
    leave
    ret
```

```
main:
    pushl %ebp
    movl %esp, %ebp
    pushl 899
    pushl 888
    pushl 877
    call foobar(int, int, int)
    addl $12, %esp
    nop
    leave
    ret
```
```assembly
; foobar(int, int, int):
  pushl %ebp
  movl %esp, %ebp
  subl $16, %esp
  movl @(%ebp), %eax
  addl $2, %eax
  movl %eax, -4(%ebp)
  movl 12(%ebp), %eax
  addl $3, %eax
  movl %eax, -8(%ebp)
  movl 16(%ebp), %eax
  addl $4, %eax
  movl %eax, -12(%ebp)
  movl -4(%ebp), %edx
  movl -8(%ebp), %eax
  addl %eax, %edx
  movl -12(%ebp), %eax
  addl %edx, %eax
  movl %eax, -16(%ebp)
  movl -4(%ebp), %eax
  imull -8(%ebp), %eax
  imull -12(%ebp), %eax
  movl %eax, %edx
  movl -16(%ebp), %eax
  addl %edx, %eax
  leave
  ret

; main:
  pushl %ebp
  movl %esp, %ebp
  pushl $99
  pushl $88
  pushl $77
  call foobar(int, int, int)
  addl $12, %esp
  nop
  leave
  ret
```
```c
foobar(int, int, int):
    pushl %ebp
    movl %esp, %ebp
    subl $16, %esp
    movl 8(%ebp), %eax
    addl $2, %eax
    movl %eax, -4(%ebp)
    movl 12(%ebp), %eax
    addl $3, %eax
    movl %eax, -8(%ebp)
    movl 16(%ebp), %eax
    addl $4, %eax
    movl %eax, -12(%ebp)
    movl -4(%ebp), %edx
    movl -8(%ebp), %eax
    addl %eax, %edx
    movl -12(%ebp), %eax
    addl %edx, %eax
    movl %eax, -16(%ebp)
    movl -4(%ebp), %eax
    imull -8(%ebp), %eax
    iaddl -12(%ebp), %eax
    movl %eax, %edx
    movl -16(%ebp), %eax
    addl %edx, %eax
    leave
    ret

main:
    pushl %ebp
    movl %esp, %ebp
    pushl $99
    pushl $88
    pushl $77
    call foobar(int, int, int)
    addl $12, %esp
    nop
    leave
    ret
```
```assembly
foobar(int, int, int):
    pushl %ebp
    movl %esp, %ebp
    subl $16, %esp
    movl 8(%ebp), %eax
    addl $2, %eax
    movl %eax, -4(%ebp)
    movl 12(%ebp), %eax
    addl $3, %eax
    movl %eax, -8(%ebp)
    movl 16(%ebp), %eax
    addl $4, %eax
    movl %eax, -12(%ebp)
    movl -4(%ebp), %edx
    movl -8(%ebp), %eax
    addl %eax, %edx
    movl -12(%ebp), %eax
    addl %edx, %eax
    movl %eax, -16(%ebp)
    movl -4(%ebp), %eax
    imull -8(%ebp), %eax
    imull -12(%ebp), %eax
    movl %eax, %edx
    movl -16(%ebp), %eax
    addl %edx, %eax
    leave
    ret

main:
    pushl %ebp
    movl %esp, %ebp
    pushl $99
    pushl $88
    pushl $77
    call foobar(int, int, int)
    addl $12, %esp
    nop
    leave
    ret
```
```c
foobar(int, int, int):
    pushl %ebp
    movl %esp, %ebp
    subl $16, %esp
    movl 8(%ebp), %eax
    addl $2, %eax
    movl %eax, -4(%ebp)
    movl 12(%ebp), %eax
    addl $3, %eax
    movl %eax, -8(%ebp)
    movl 16(%ebp), %eax
    addl $4, %eax
    movl %eax, -12(%ebp)
    movl -4(%ebp), %edx
    movl -8(%ebp), %eax
    imull -8(%ebp), %eax
    imull -12(%ebp), %eax
    movl %eax, %edx
    movl -16(%ebp), %eax
    addl %edx, %eax
    leave
    ret
```

```
main:
    pushl %ebp
    movl %esp, %ebp
    pushl $99
    pushl $88
    pushl $77
    call foobar(int, int, int)
    addl $12, %esp
    nop
    leave
    ret
```
```c
foobar(int, int, int):
    pushl %ebp
    movl %esp, %ebp
    subl $16, %esp
    movl 8(%ebp), %eax
    addl $2, %eax
    movl %eax, -4(%ebp)
    movl 12(%ebp), %eax
    addl $3, %eax
    movl %eax, -8(%ebp)
    movl 16(%ebp), %eax
    addl $4, %eax
    movl %eax, -12(%ebp)
    movl -4(%ebp), %edx
    movl -8(%ebp), %eax
    addl %eax, %edx
    movl -12(%ebp), %eax
    addl %edx, %eax
    movl %eax, -16(%ebp)
    imull -8(%ebp), %eax
    imull -12(%ebp), %eax
    movl %eax, %edx
    movl -16(%ebp), %eax
    addl %edx, %eax
    leave
    ret
```

```
main:
    pushl %ebp
    movl %esp, %ebp
    pushl $99
    pushl $88
    pushl $77
    call foobar(int, int, int)
    addl $12, %esp
    nop
    leave
    ret
```
fooobar(int, int, int):

pushl %ebp
movl %esp, %ebp
subl $16, %esp
movl 8(%ebp), %eax
addl $2, %eax
movl %eax, -4(%ebp)
movl 12(%ebp), %eax
addl $3, %eax
movl %eax, -8(%ebp)
movl 16(%ebp), %eax
addl $4, %eax
movl %eax, -12(%ebp)
movl -4(%ebp), %edx
addl %eax, %edx
movl -12(%ebp), %eax
addl %edx, %eax
movl %eax, -16(%ebp)
imull -8(%ebp), %eax
imull -12(%ebp), %eax
movl %eax, %edx
movl -16(%ebp), %eax
addl %edx, %eax
leave
ret

main:
pushl %ebp
movl %esp, %ebp
pushl $99
pushl $88
pushl $77
call fooobar(int, int, int)
addl $12, %esp
nop
leave
ret

%ebp

%esp

$99
$88
$77
$79
$91
$103
$293

0xfffffd0d8

0x08049bbc

0xfffffd0d8
fooobar(int, int, int):

1. pushl %ebp
2. movl %esp, %ebp
3. subl $16, %esp
4. movl 8(%ebp), %eax
5. addl $2, %eax
6. movl %eax, -4(%ebp)
7. movl 12(%ebp), %eax
8. addl $3, %eax
9. movl %eax, -8(%ebp)
10. movl 16(%ebp), %eax
11. addl $4, %eax
12. movl %eax, -12(%ebp)
13. movl -4(%ebp), %edx
14. movl -8(%ebp), %eax
15. addl %edx, %edx
16. movl -12(%ebp), %eax
17. addl %edx, %eax
18. movl %eax, -16(%ebp)
19. movl -4(%ebp), %eax
20. imull -8(%ebp), %eax
21. imull -12(%ebp), %eax
22. movl %eax, %edx
23. movl -16(%ebp), %eax
24. addl %edx, %eax
25. leave
26. ret

main:

29. pushl %ebp
30. movl %esp, %ebp
31. pushl $99
32. pushl $88
33. pushl $77
34. call fooobar(int, int, int)
35. addl $12, %esp
36. nop
37. leave
38. ret

$99
$88
$77

0x08049bbc

0xffffd0d8

%esp, %ebp

$79
$91
$103
$293

0xfffffd0d8
```
foo((int, int, int):
  pushl %ebp
  movl %esp, %ebp
  subl $16, %esp
  movl 0(%ebp), %eax
  addl $2, %eax
  movl %eax, -4(%ebp)
  movl 12(%ebp), %eax
  addl $3, %eax
  movl %eax, -8(%ebp)
  movl 16(%ebp), %eax
  addl $4, %eax
  movl %eax, -12(%ebp)
  movl -4(%ebp), %edx
  addl %eax, %edx
  movl -12(%ebp), %eax
  addl %edx, %eax
  movl %eax, -16(%ebp)
  movl -4(%ebp), %eax
  imull -8(%ebp), %eax
  imull -12(%ebp), %eax
  movl %eax, %edx
  movl -16(%ebp), %eax
  addl %edx, %eax
  leave
  ret

main:
  pushl %ebp
  movl %esp, %ebp
  pushl $99
  pushl $88
  pushl $77
  call foo((int, int, int)
  addl $12, %esp
  nop
  leave
  ret
```
```c
fooobar(int, int, int):
    pushl %ebp
    movl %esp, %ebp
    subl $16, %esp
    movl 8(%ebp), %eax
    addl $2, %eax
    movl %eax, -4(%ebp)
    movl 12(%ebp), %eax
    addl $3, %eax
    movl %eax, -8(%ebp)
    movl 16(%ebp), %eax
    addl $4, %eax
    movl %eax, -12(%ebp)
    movl -4(%ebp), %edx
    movl -8(%ebp), %eax
    imull -8(%ebp), %eax
    imull -12(%ebp), %eax
    movl %eax, %edx
    movl -16(%ebp), %eax
    addl %edx, %eax
    leave
    ret

main:
    pushl %ebp
    movl %esp, %ebp
    pushl $99
    pushl $88
    pushl $77
    call foobar(int, int, int)
    addl $12, %esp
    nop
    leave
    ret
```

%esp, %ebp → $99
$88
$77
0x08049bbc
0xffffffffd0d8
%eip = 0x08049bbc
Example 1

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

int main(int argc, char**argv) {
    char nice[] = "is nice."
    char name[8];
    gets(name);
    printf("%s %s\n", name, nice);
    return 0;
}
```
```c
#include <stdio.h>
#include <string.h>

int main(int argc, char**argv) {
    char nice[] = "is nice."
    char name[8];
    gets(name);
    printf("%s %s\n", name, nice);
    return 0;
}
```
Example 1

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

int main(int argc, char**argv) {
    char nice[] = "is nice."
    char name[8];
    gets(name);
    printf("%s %s\n",name,nice);
    return 0;
}
```
#include <stdio.h>
#include <string.h>

int main(int argc, char**argv) {
    char nice[] = "is nice."
    char name[8];
    gets(name);
    printf("%s %s\n",name,nice);
    return 0;
}
Example 1

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

int main(int argc, char**argv) {
    char nice[] = "is nice."
    char name[8];
    gets(name);
    printf("%s %s\n",name,nice);
    return 0;
}
```

What happens if we read a long name?
Example 1

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

int main(int argc, char**argv) {
    char nice[] = "is nice."
    char name[8];
    gets(name);
    printf("%s %s\n",name,nice);
    return 0;
}
```

What happens if we read a long name?
Example 1

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

int main(int argc, char**argv) {
    char nice[] = "is nice."
    char name[8];
    gets(name);
    printf("%s %s\n",name,nice);
    return 0;
}
```

If not null terminated can read more of the stack
Let's run this program!
```c
#include <stdio.h>
#include <string.h>

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xdeadbeef;
    char buf[4];
    strcpy(buf,str);
}

int main(int argc, char**argv) {
    func(0xffffffff,0xaaaaaaaa,argv[1]);
    return 0;
}
```
```c
#include <stdio.h>
#include <string.h>

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xdeadbeef;
    char buf[4];
    strcpy(buf,str);
    
    main(int argc, char**argv) {
        func(0xaaaaaaaa,0xbbbbbbbb,argv[1]);
        return 0;
    }
```
Example 2

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

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xdeadbeef;
    char buf[4];
    strcpy(buf,str);
}

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

<table>
<thead>
<tr>
<th>argv[1]</th>
<th>0xbbbbbbbbbb</th>
</tr>
</thead>
<tbody>
<tr>
<td>saved ret</td>
<td>0xaaaaaaaaaa</td>
</tr>
<tr>
<td>saved ebp</td>
<td>0xdeadbeef</td>
</tr>
<tr>
<td>buf[0-3]</td>
<td></td>
</tr>
</tbody>
</table>
```c
#include <stdio.h>
#include <string.h>

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xdeadbeef;
    char buf[4];
    strcpy(buf,str);
}

int main(int argc, char**argv) {
    func(0xffffffff,0xbbbbbbbbb,argv[1]);
    return 0;
}
```
Example 2

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

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xdeadbeef;
    char buf[4];
    strcpy(buf,str);
}

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

Example 2

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

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xdeadbeef;
    char buf[4];
    strcpy(buf, str);
}

int main(int argc, char**argv) {
    func(0xaaaaaaaa, 0xbbbbbbbbb, argv[1]);
    return 0;
}
```
Example 2

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

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xdeadbeef;
    char buf[4];
    strcpy(buf,str);
}

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

If first argument to program is “AAAAAAAAAAAAA...”
```c
#include <stdio.h>
#include <string.h>

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xdeadbeef;
    char buf[4];
    strcpy(buf,str);
}

int main(int argc, char**argv) {
    func(0xaaaaaaaa, 0xbbbbbbbb, argv[1]);
    return 0;
}
```
Example 2

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

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xdeadbeef;
    char buf[4];
    strcpy(buf,str);
}

int main(int argc, char**argv) {
    func(0xaaaaaaaa,0xbbbbbbbbb,argv[1]);
    return 0;
}
```
Example 2

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

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xdeadbeef;
    char buf[4];
    strcpy(buf,str);
}

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

%esp →

%ebp = 0x41414141
#include <stdio.h>
#include <string.h>

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xdeadbeef;
    char buf[4];
    strcpy(buf,str);
}

int main(int argc, char**argv) {
    func(0xaaaaaaaa,0xbbbbbbbb,argv[1]);
    return 0;
}
```c
#include <stdio.h>
#include <string.h>

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xdeadbeef;
    char buf[4];
    strcpy(buf, str);
}

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

%eip = 0x41414141
%ebp = 0x41414141

%eip = 0x41414141
%esp
%ebp = 0x41414141

%eip = 0x41414141
Stack Buffer Overflow

- If source string of `strcpy` controlled by attacker (and destination is on the stack)
  - Attacker gets to control where the function returns by overwriting the return address
  - Attacker gets to transfer control to anywhere!
- Where do you jump?
Existing functions

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

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xdeadbeef;
    char buf[4];
    strcpy(buf,str);
}

int main(int argc, char**argv) {
    func(0xaaaaaaaa,0xbbbbbbbbbb,argv[1]);
    return 0;
}
```
**Existing functions**

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

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xdeadbeef;
    char buf[4];
    strcpy(buf, str);
}

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

%ebp  %esp, %ebp →

<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x41414141</td>
</tr>
<tr>
<td>0x41414141</td>
</tr>
<tr>
<td>0x41414141</td>
</tr>
<tr>
<td>0x41414141</td>
</tr>
<tr>
<td>0x41414141</td>
</tr>
<tr>
<td>0x41414141</td>
</tr>
<tr>
<td>0x41414141</td>
</tr>
<tr>
<td>0x41414141</td>
</tr>
</tbody>
</table>
Existing functions

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

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xdeadbeef;
    char buf[4];
    strcpy(buf,str);
}

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

%ebp = 0x41414141

%esp →

```
0x41414141
0x41414141
0x41414141
0x41414141
0x08049b95
0x41414141
0x41414141
0x41414141
```

%esp →

```
0x41414141
0x41414141
0x41414141
0x41414141
```
#include <stdio.h>
#include <string.h>

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xdeadbeef;
    char buf[4];
    strcpy(buf,str);
}

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

%esp = 0x41414141
%ebp = 0x41414141
%eip = 0x08049b95
Existing functions

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

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xdeadbeef;
    char buf[4];
    strcpy(buf, str);
}

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

%eip = 0x08049b95
%esp = 0x41414141
%ebp = 0x41414141
Let’s look at this in GDB (w/ GEF)
Better Hijacking Control

• Jump to attacker-supplied code

• Where? We have control of string!
  ➤ Put code in string
  ➤ Jump to start of string

<table>
<thead>
<tr>
<th>%ebp</th>
<th>argv[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xbbbbbbbbbb</td>
<td>0xaaaaaaaaa</td>
</tr>
<tr>
<td>saved ret</td>
<td>saved ebp</td>
</tr>
<tr>
<td>0xdeadbeef</td>
<td>buf[0-3]</td>
</tr>
</tbody>
</table>
Better Hijacking Control

- Jump to attacker-supplied code
- Where? We have control of string!
  - Put code in string
  - Jump to start of string
Shellcode

- **Shellcode:** small code fragment that receives initial control in an control flow hijack exploit
  - Control flow hijack: taking control of instruction ptr
- Earliest attacks used shellcode to exec a shell
  - Target a setuid root program, gives you root shell
void main() {
  char *name[2];

  name[0] = "/bin/sh";
  name[1] = NULL;
  execve(name[0], name, NULL);
}

Shellcode
Shellcode

- Can we just take output from gcc/clang?
Shellcode

- There are some restrictions
  1. Shellcode cannot contain null characters ‘\0’
     - Why not?
     - Fix: use different instructions and NOPs to eliminate \0
  2. If payload is via gets() must also avoid line-breaks
How do we make this robust?

- 3. Exact address of shellcode start not always easy to guess
  - Miss? SEGFAULT!
- Fix: NOP-sled
How do we make this robust?

- 3. Exact address of shellcode start not always easy to guess
  - Miss? SEGFAULT!
- Fix: NOP-sled
How do we make this robust?

- 3. Exact address of shellcode start not always easy to guess
  - Miss? SEGFAULT!
- Fix: NOP-sled
Metasploit helps!

msf payload(`shell_bind_tcp`) > generate -h
Usage: generate [options]

Generates a payload.

OPTIONS:

- `E` Force encoding.
- `b` The list of characters to avoid: '\x00\xff'
- `e` The name of the encoder module to use.
- `f` The output file name (otherwise stdout).
- `h` Help banner.
- `i` the number of encoding iterations.
- `k` Keep the template executable functional.
- `o` A comma separated list of options in VAR=VAL format.
- `p` The Platform for output.
- `s` NOP sled length.
- `t` The output format: `raw,ruby,rb,perl,pl,c,js_be,js_le,java,dll,exe,exe-small,elf,macho,vba,t`.
- `x` The executable template to use.
Buffer Overflow Defenses

- Avoid unsafe functions
- Stack canary
- Separate control stack
- Address Space Layout Randomization (ASLR)
- Memory writable or executable, not both ($W^X$)
- Control flow integrity (CFI)
Avoiding Unsafe Functions

• strcpy, strcat, gets, etc.

• **Plus:** Good idea in general

• **Minus:** Requires manual code rewrite

• **Minus:** Non-library functions may be vulnerable
  ➢ E.g. user creates their own strcpy

• **Minus:** No guarantee you found everything

• **Minus:** alternatives are also error-prone
Even printf is tricky

If buf is under control of attacker is: printf("%s\n", buf) safe?
Even printf is tricky

If buf is under control of attacker is: printf(buf) safe?
Even printf is tricky

Is printf(“%s\n”) safe?
Even printf is tricky

printf can be used to read and write memory

control flow hijacking!

Exploiting Format String Vulnerabilities

scut / team teso

September 1, 2001

https://crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf
Buffer Overflow Defenses

- Avoid unsafe functions
- Stack canary
- Separate control stack
- Address Space Layout Randomization (ASLR)
- Memory writable or executable, not both ($W^X$)
- Control flow integrity (CFI)
Stack Canary

• Special value placed before return address
  ➢ Secret random value chosen at program start
  ➢ String terminator ‘\0’

• Gets overwritten during buffer overflow

• Check canary before jumping to return address

• Automatically inserted by compiler
  ➢ GCC: -fstack-protector or -fstack-protector-strong
Back to Example 2

```
#include <stdio.h>
#include <string.h>

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xdeadbeef;
    char buf[4];
    strcpy(buf,str);
}

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

Check canary on ret
Stack Canary

- **Plus**: No code changes required, only recompile
- **Minus**: Performance penalty per return
- **Minus**: Only protects against stack smashing
- **Minus**: Fails if attacker can read memory
Buffer Overflow Defenses

- Avoid unsafe functions
- Stack canary
- Separate control stack
- Address Space Layout Randomization (ASLR)
- Memory writable or executable, not both (W^X)
- Control flow integrity (CFI)
Separate Stack

“SafeStack is an instrumentation pass that protects programs against attacks based on stack buffer overflows, without introducing any measurable performance overhead. It works by separating the program stack into two distinct regions: the safe stack and the unsafe stack. The safe stack stores return addresses, register spills, and local variables that are always accessed in a safe way, while the unsafe stack stores everything else. This separation ensures that buffer overflows on the unsafe stack cannot be used to overwrite anything on the safe stack.”

WebAssembly has separate stack (kind of)!
Address Space Layout Randomization

- Change location of stack, heap, code, static vars
- Works because attacker needs address of shellcode
- Layout must be unknown to attacker
  - Randomize on every launch (best)
  - Randomize at compile time
- Implemented on most modern OSes in some form
Traditional Memory Layout

Stack

mapped

heap

.bss

.data

.text
PaX Memory Layout

random stack base → Stack

random base → mapped

random base → heap

random base → .bss

random base → .data

random base → .text
32-bit PaX ASLR (x86)

Stack:

```
1 0 1 0
fixed
random (24 bits)
zero
```

Mapped area:

```
0 1 0 0
fixed
random (16 bits)
zero
```

Executable code, static variables, and heap:

```
0 0 0 0
fixed
random (16 bits)
zero
```
**Address Space Layout Randomization**

- **Plus:** No code changes or recompile required
- **Minus:** 32-bit arch get limited protection
- **Minus:** Fails if attacker can read memory
- **Minus:** Load-time overhead
- **Minus:** No exec img sharing between processes
W^X: write XOR execute

- Use MMU to ensure memory cannot be both writeable and executable at same time
- Code segment: executable, not writeable
- Stack, heap, static vars: writeable, not executable
- Supported by most modern processors
- Implemented by modern operating systems
W^X: write XOR execute

• **Plus:** No code changes or recompile required
• **Minus:** Requires hardware support
• **Minus:** Defeated by return-oriented programming
• **Minus:** Does not protect JITed code
Buffer Overflow Defenses

- Avoid unsafe functions
- Stack canary
- Separate control stack
- Address Space Layout Randomization (ASLR)
- Memory writable or executable, not both (W^X)
- Control flow integrity (CFI)
Control Flow Integrity

- Check destination of every indirect jump
  - Function returns
  - Function pointers
  - Virtual methods

- What are the valid destinations?
  - Caller of every function known at compile time
  - Class hierarchy limits possible virtual function instances
CFI

- **Plus:** No code changes or hardware support
- **Plus:** Protects against many vulnerabilities
- **Minus:** Performance overhead
- **Minus:** Requires smarter compiler
- **Minus:** Requires having all code available