CSE 127 Computer Security

~~Deian Stefan~~,
Stefan Savage, Winter 2018, Lecture 3

Low Level Software Security I: Buffer Overflows and Stack Smashing

When is a program secure?

- When it does exactly what it should?
  - Not more.
  - Not less.

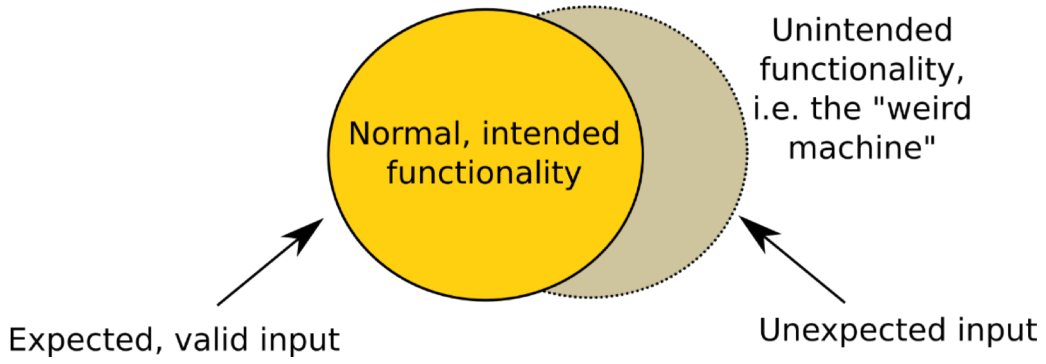
- But how do we know what a program is supposed to do?
  - Somebody tells us? (But do we trust them?)
  - We write the code ourselves?
    (But what fraction of the software you use have you written?)

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"
- 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


Normal, intended functionality
Unintended functionality, i.e. the "weird machine"
Expected, valid input
Unexpected input

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 the 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
- Correctly fix buffer overflow vulnerabilities

Buffer Overflow

- ***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

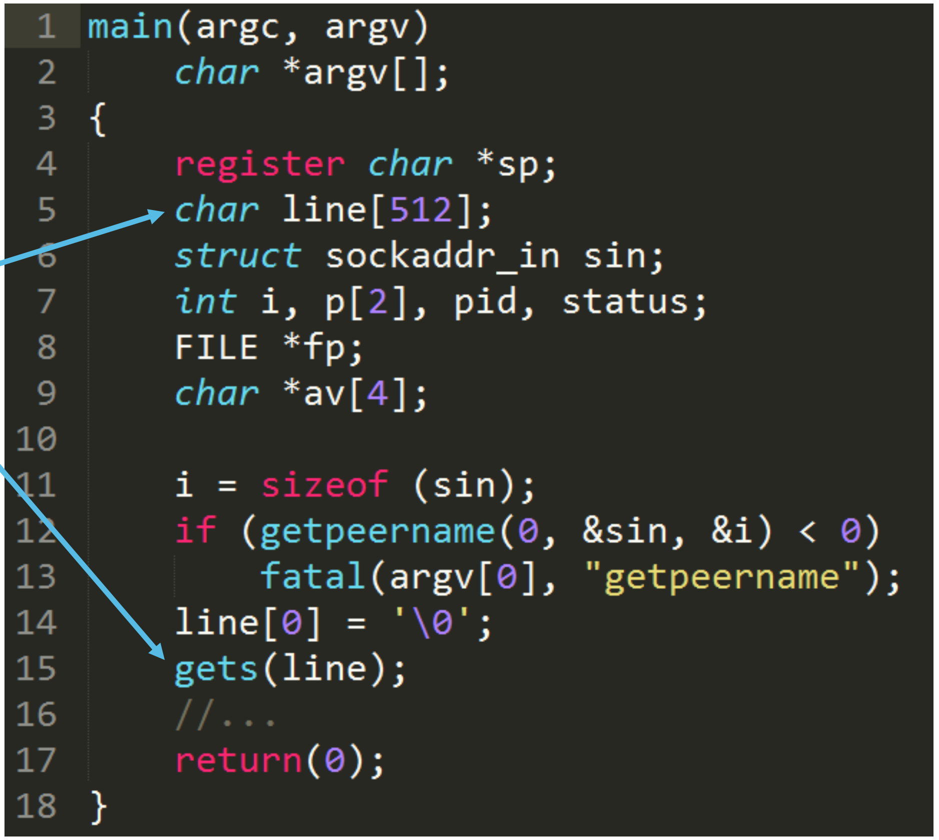
Buffer Overflow

- 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.
- 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

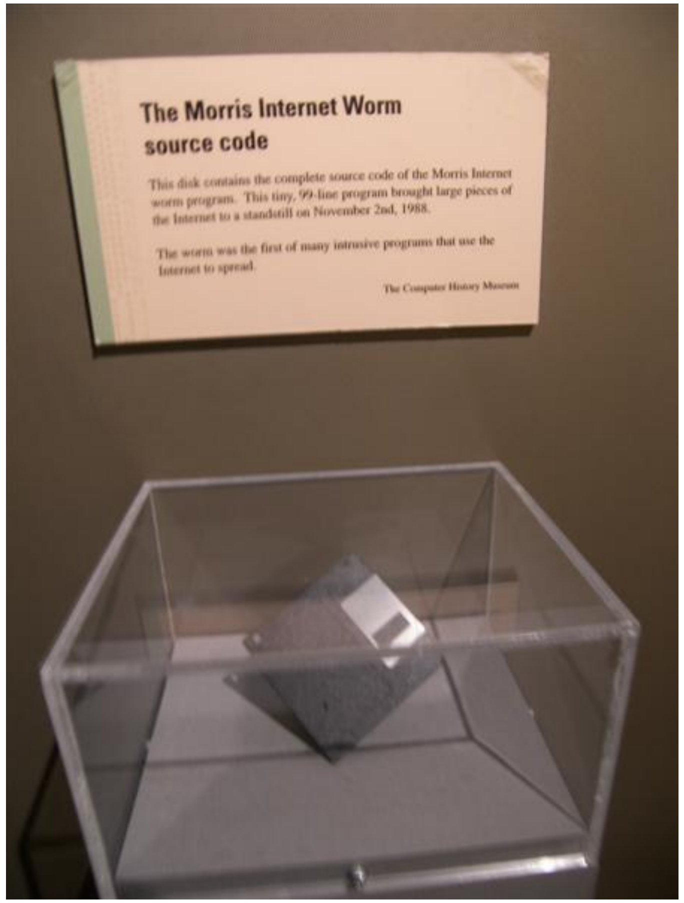


```
main(argc, argv)
	char *argv[];
{
	register char *sp;
	char line[512];
	struct sockaddr_in sin;
	int i, p[2], pid, status;
	FILE *fp;
	char *av[4];

	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

Morris Worm

- This fingerd 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 Computer Fraud and Abuse Act (CFAA)


The Morris Internet Worm
source code

This disk contains the complete source code of the Morris Internet worm program. This tiny, 99-line program brought large pieces of the Internet to a standstill on November 2nd, 1988.

The worm was the first of many intrusive programs that use the Internet to spread.

The Computer History Museum

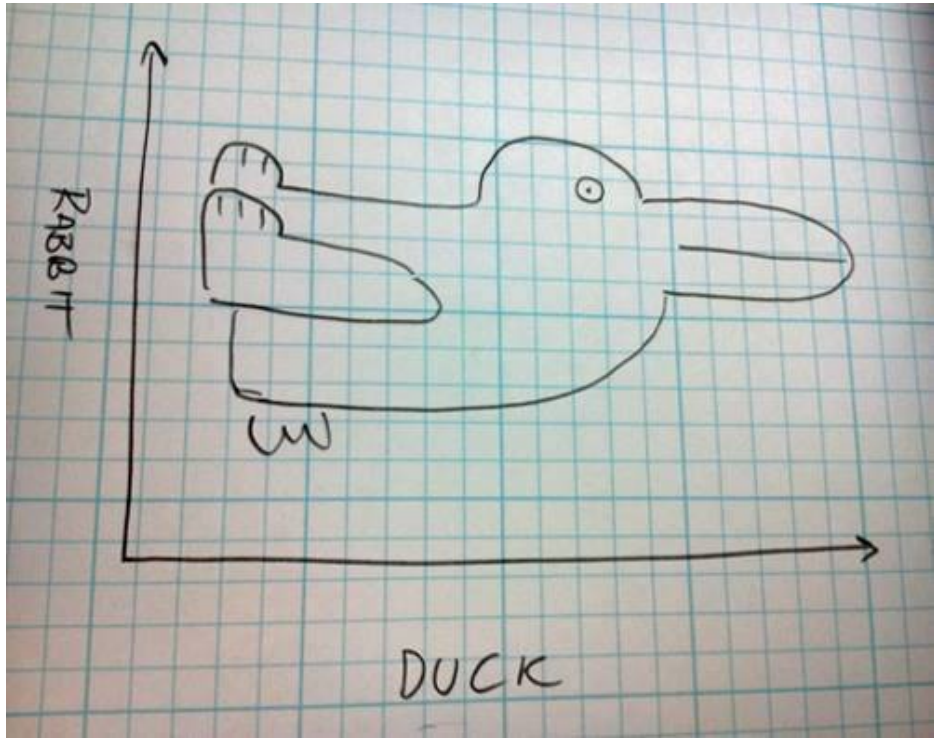
https://en.wikipedia.org/wiki/Morris_worm

Ok but…

- Why does overflowing a buffer let you take over the machine?
- That seems crazy no?

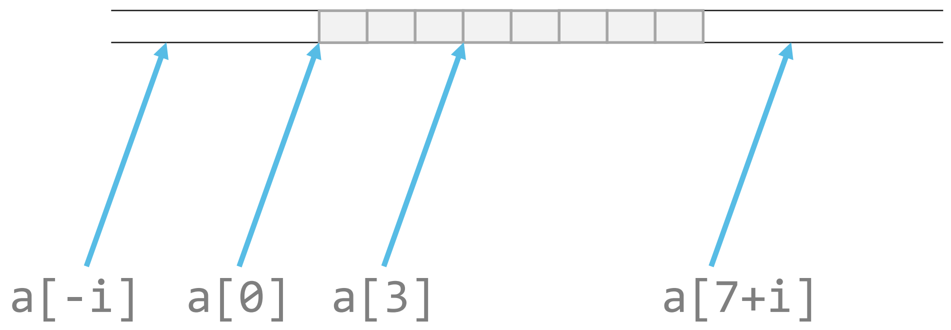
Changing Perspectives

- Your program manipulates data
- Data manipulates your program



Buffer Overflow

- 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 spec say?
    - What happens in most implementations?



Understanding Function Calls

- How does a function call work?
  - What's the abstraction?

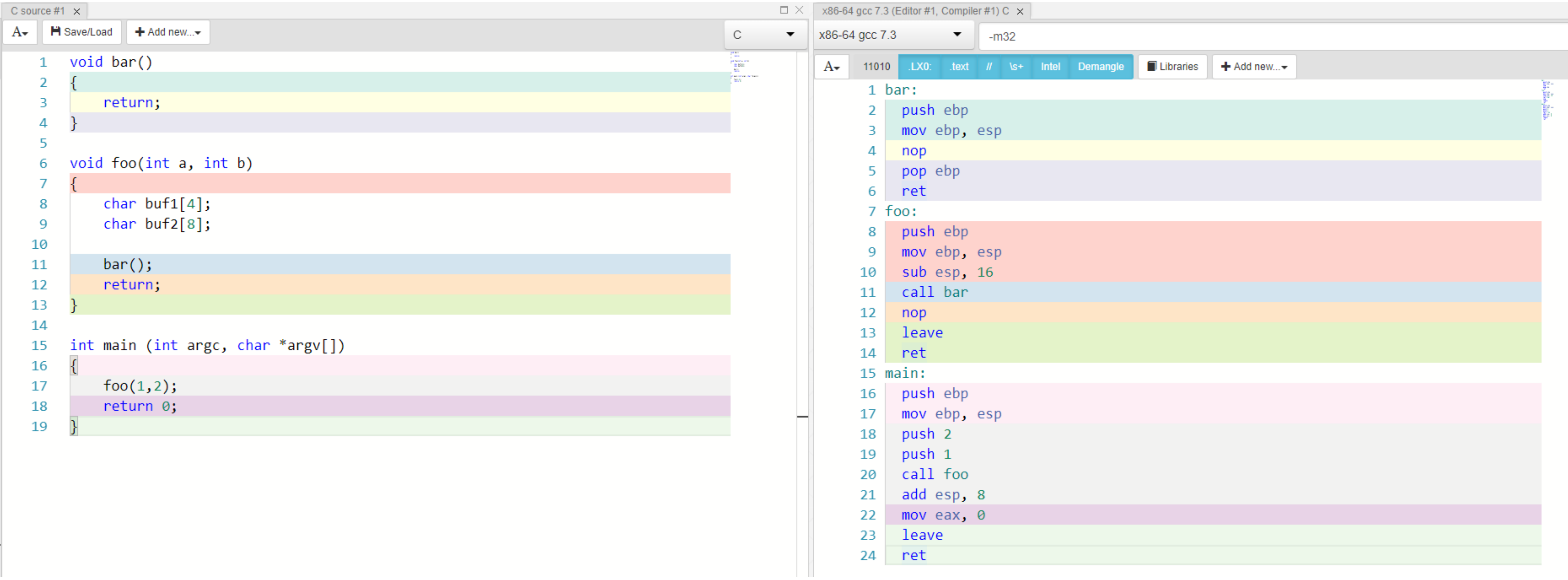
    ```
    …
    foo();
    …
    ```

  - What's the reality?
    - How does the called function know where to return to?
    - Where is the return address stored?

```
void foo()
{
    …
    …
    return;
}
```

Understanding Function Calls

- godbolt compiler explorer: https://godbolt.org/

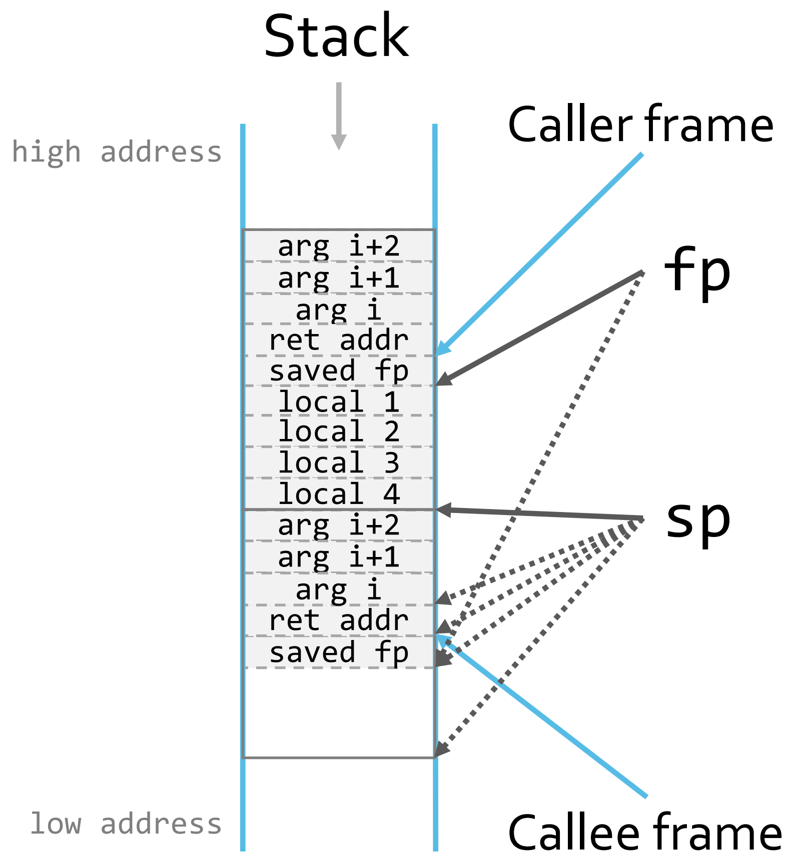


```c
void bar()
{
    return;
}

void foo(int a, int b)
{
    char buf1[4];
    char buf2[8];

    bar();
    return;
}

int main (int argc, char *argv[])
{
    foo(1,2);
    return 0;
}
```

```
bar:
  push ebp
  mov ebp, esp
  nop
  pop ebp
  ret
foo:
  push ebp
  mov ebp, esp
  sub esp, 16
  call bar
  nop
  leave
  ret
main:
  push ebp
  mov ebp, esp
  push 2
  push 1
  call foo
  add esp, 8
  mov eax, 0
  leave
  ret
```

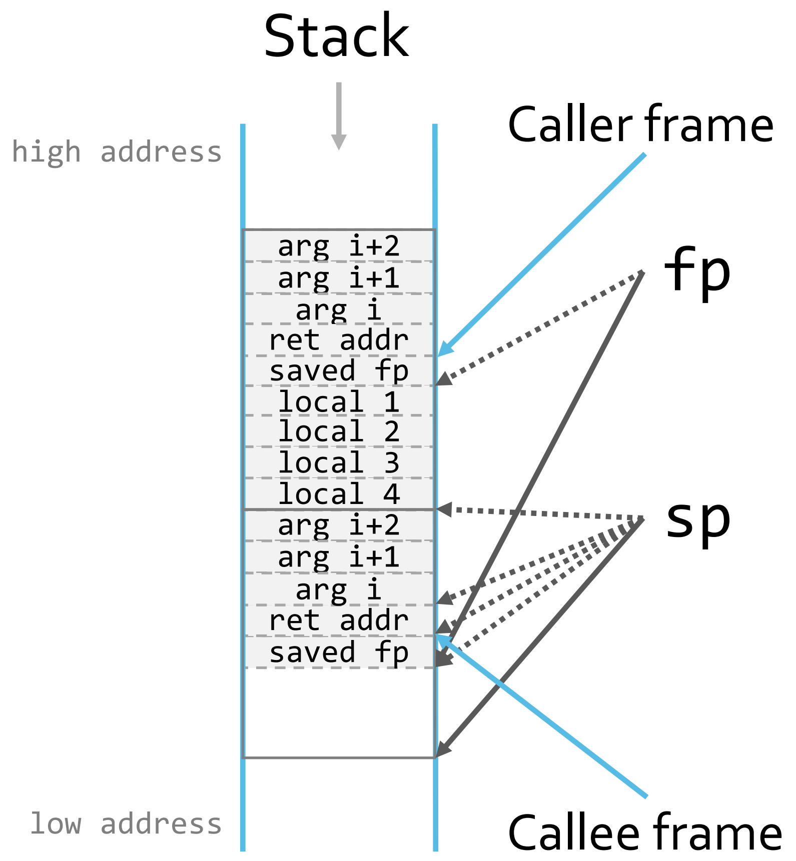
Understanding Function Calls

- 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)



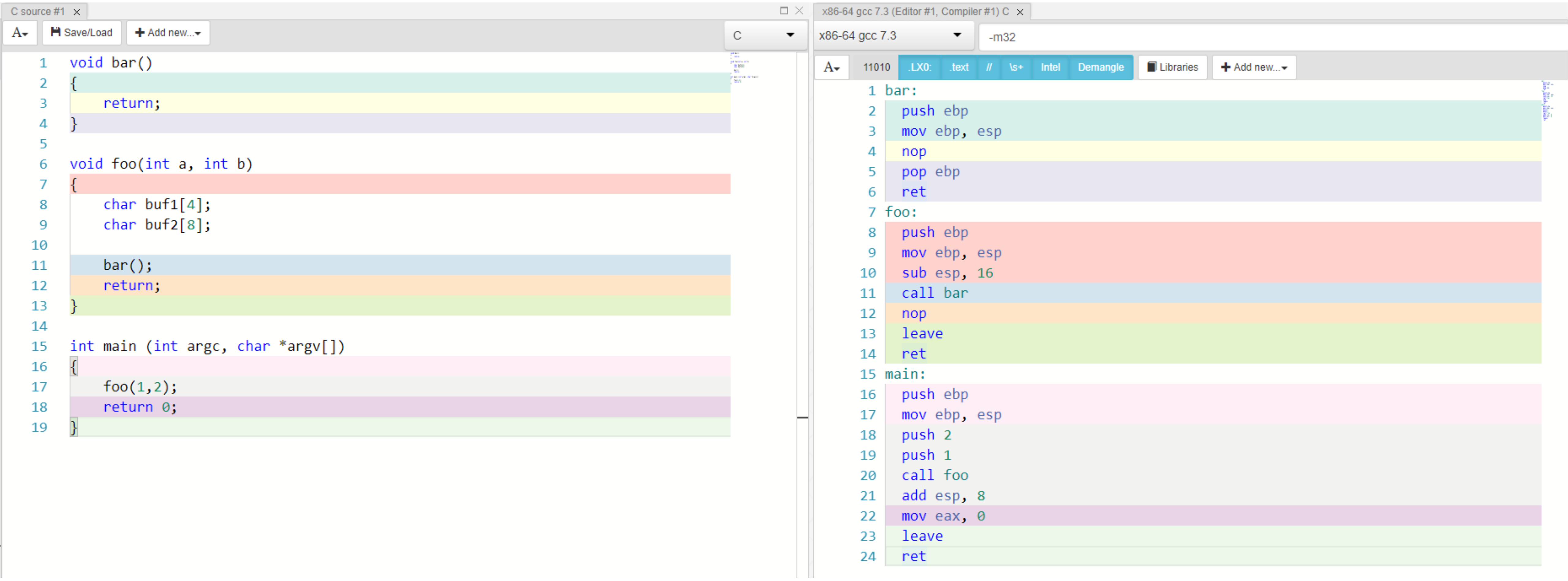
Understanding Function Calls

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



Understanding Function Calls

- godbolt compiler explorer: https://godbolt.org/

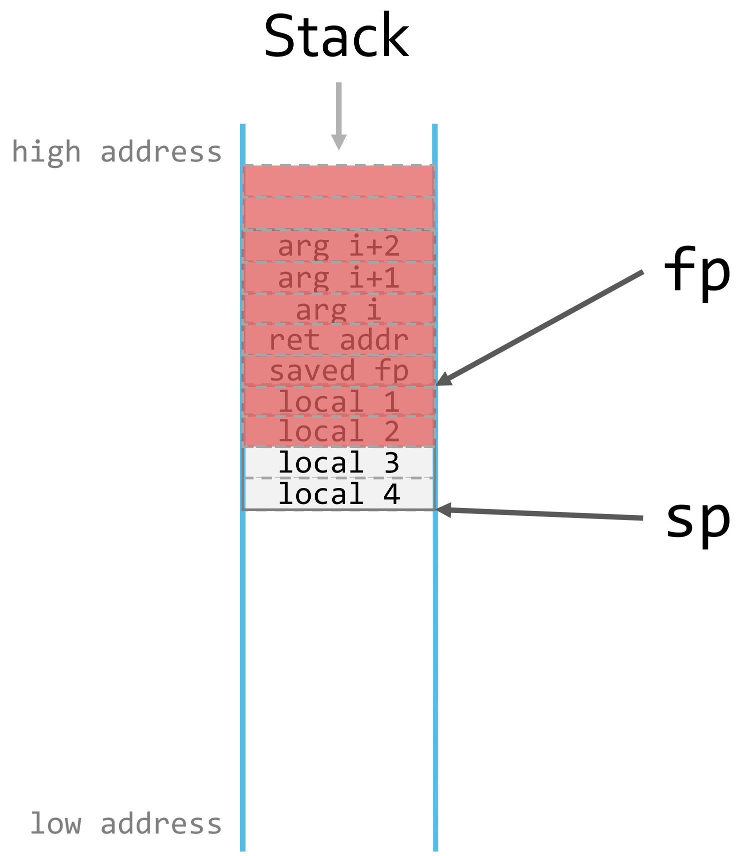


```c
void bar()
{
    return;
}

void foo(int a, int b)
{
    char buf1[4];
    char buf2[8];

    bar();
    return;
}

int main (int argc, char *argv[])
{
    foo(1,2);
    return 0;
}
```

```asm
bar:
  push ebp
  mov ebp, esp
  nop
  pop ebp
  ret
foo:
  push ebp
  mov ebp, esp
  sub esp, 16
  call bar
  nop
  leave
  ret
main:
  push ebp
  mov ebp, esp
  push 2
  push 1
  call foo
  add esp, 8
  mov eax, 0
  leave
  ret
```

Smashing The Stack

- Mixing control and user data is never a good idea.
- What happens if you overwrite an attacker-supplied value past the bounds of a local variable?
  - Let's say we overflow `local 3`
- Overwriting
  - Another local variable
  - Saved frame pointer
  - Return address
  - Function arguments
  - Deeper stack frames
    - Overwrite often happens outside of current function's frame
  - Exception control data

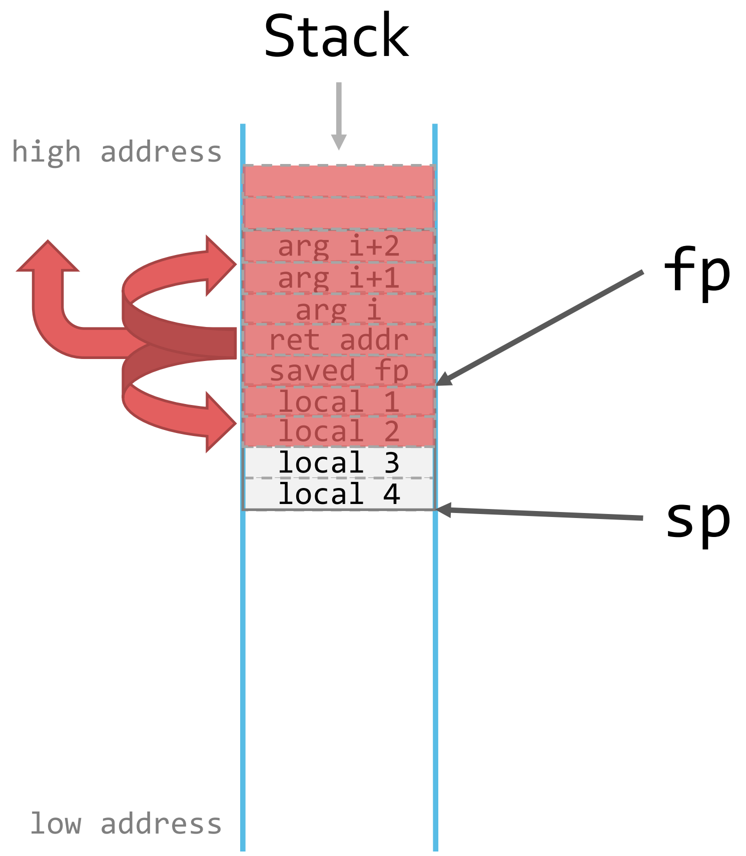


Smashing The Stack

- 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

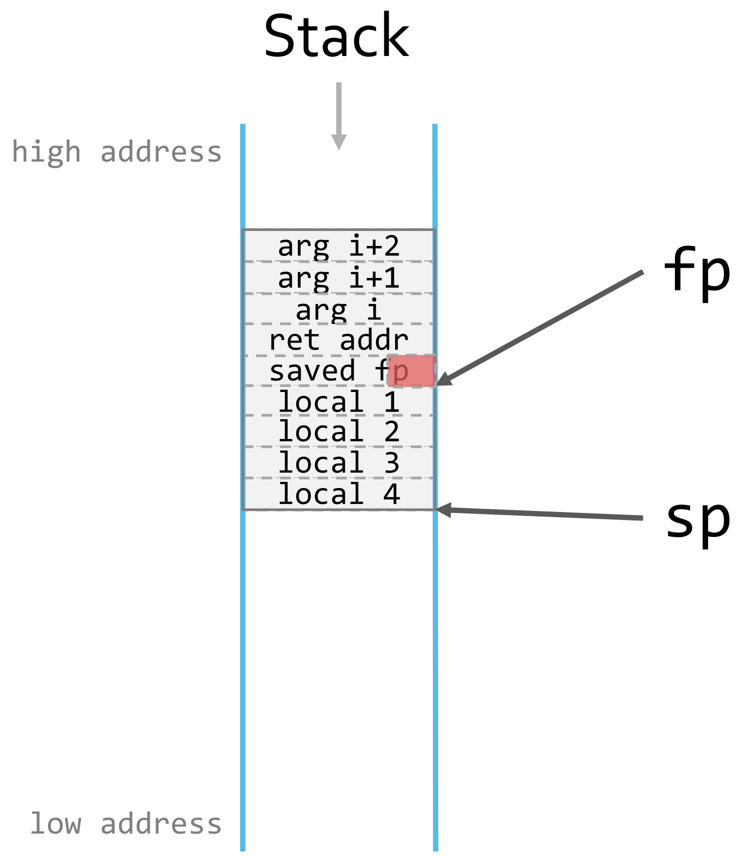
Smashing The Stack

- 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, or code that already exists in the process
      - More on this later
    - Game over



Smashing The Stack

- Overwriting the saved frame pointer
  - Upon function return, stack moves to an attacker-supplied address
  - Control of the stack leads to control of execution
  - Even a single byte may be enough!

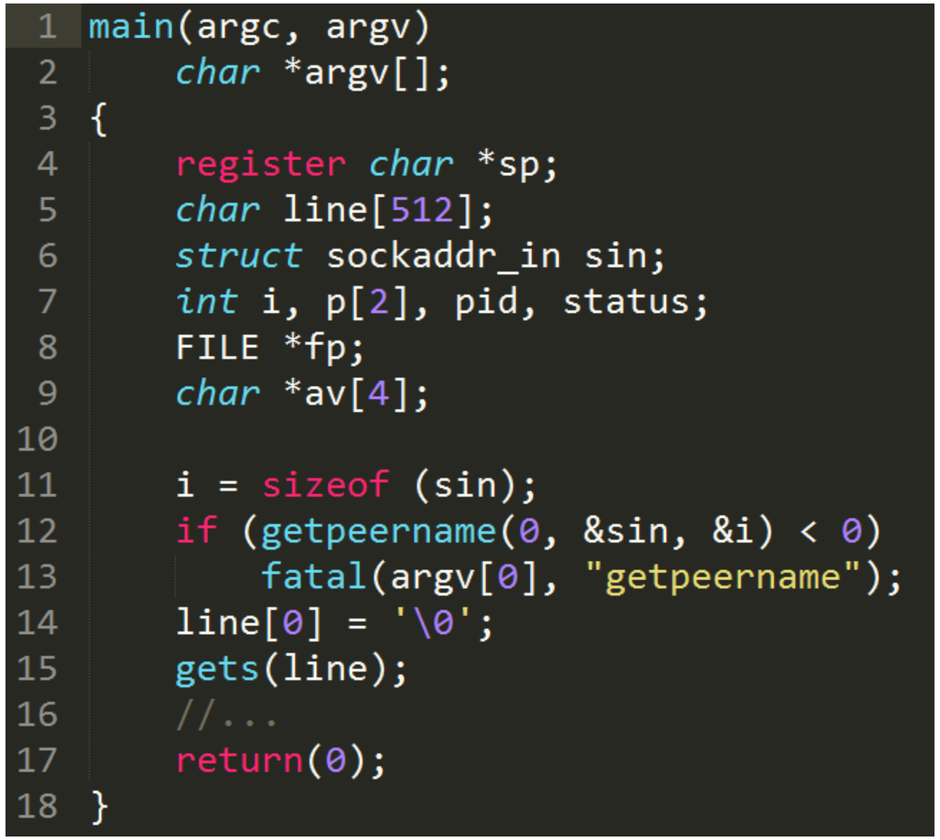


Buffer Overflow Patterns

- Spotting buffer overflow bugs in code
  - Missing Check
  - Avoidable Check
  - Wrong Check

Buffer Overflow Code Patterns

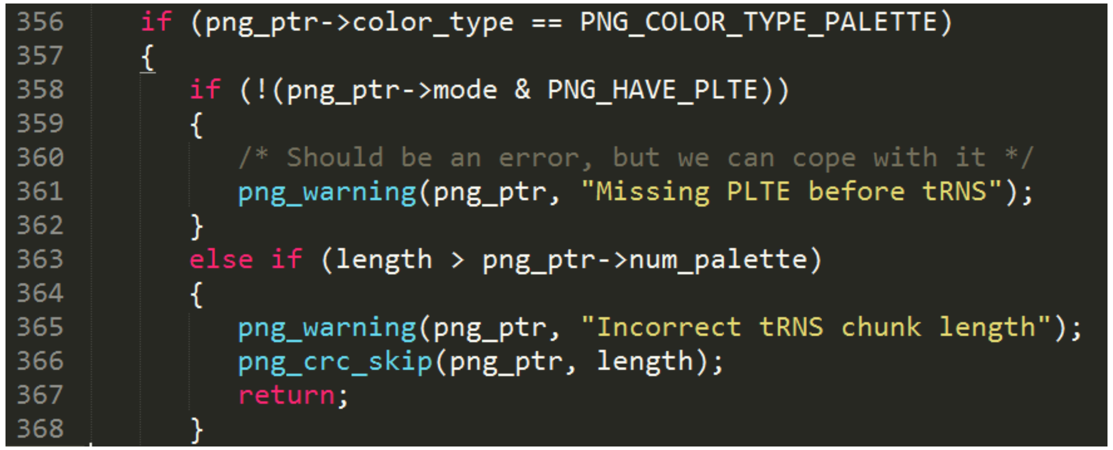
- Missing Check
  - No test to make sure memory writes stay within intended bounds
- Example
  - fingerd



```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);
}
```

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

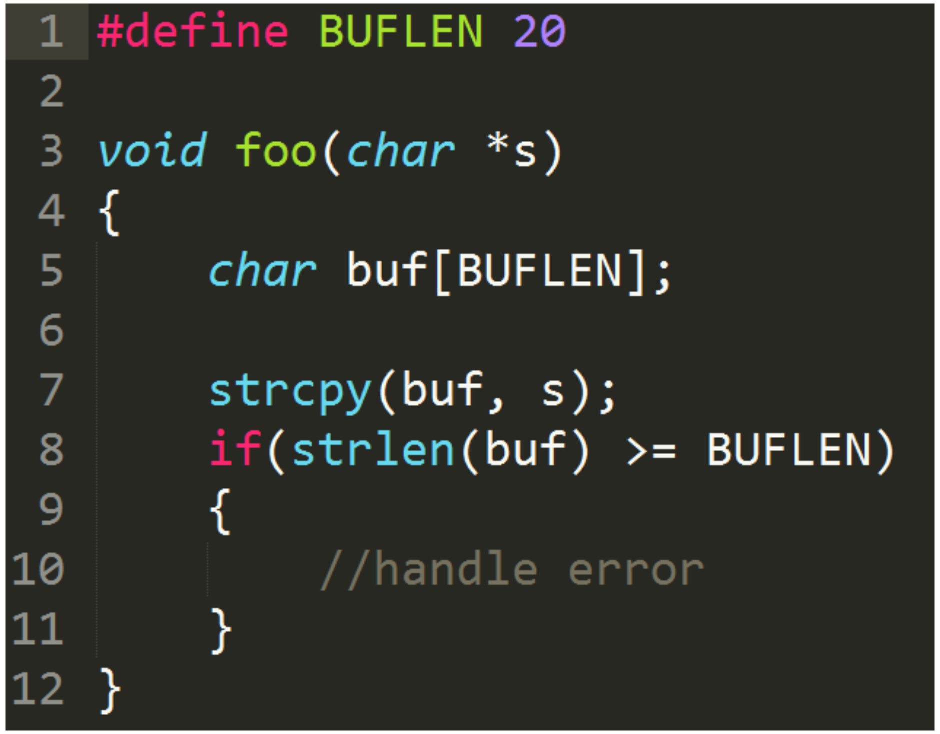


```
    if (png_ptr->color_type == PNG_COLOR_TYPE_PALETTE)
    {
        if (!(png_ptr->mode & PNG_HAVE_PLTE))
        {
            /* Should be an error, but we can cope with it */
            png_warning(png_ptr, "Missing PLTE before tRNS");
        }
        else if (length > png_ptr->num_palette)
        {
            png_warning(png_ptr, "Incorrect tRNS chunk length");
            png_crc_skip(png_ptr, length);
            return;
        }
```

http://www.libpng.org/pub/png/libpng.html

Buffer Overflow Code Patterns

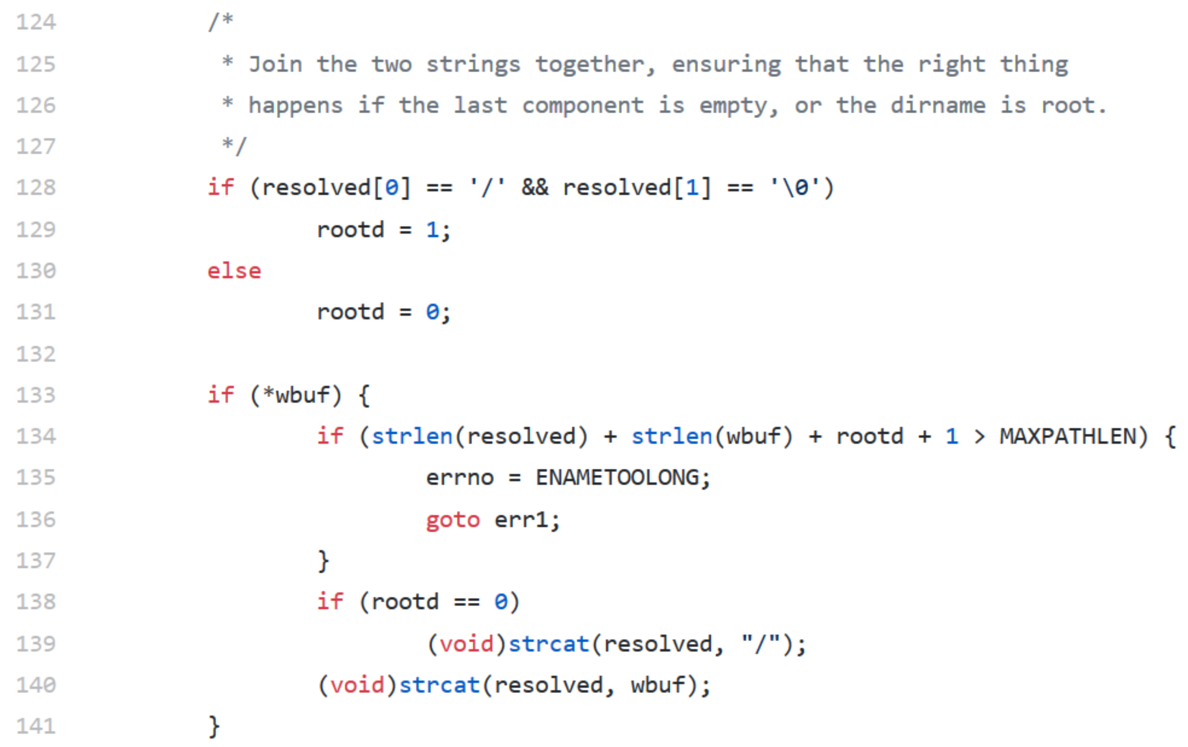
- Avoidable Check
  - Special case: check is late
  - There is a test to make sure memory writes stay within intended bounds, but it is placed after the offending operation



```c
#define BUFLEN 20

void foo(char *s)
{
    char buf[BUFLEN];

    strcpy(buf, s);
    if(strlen(buf) >= BUFLEN)
    {
        //handle error
    }
}
```

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



```c
/*
 * Join the two strings together, ensuring that the right thing
 * happens if the last component is empty, or the dirname is root.
 */
if (resolved[0] == '/' && resolved[1] == '\0')
	rootd = 1;
else
	rootd = 0;

if (*wbuf) {
	if (strlen(resolved) + strlen(wbuf) + rootd + 1 > MAXPATHLEN) {
		errno = ENAMETOOLONG;
		goto err1;
	}
	if (rootd == 0)
		(void)strcat(resolved, "/");
	(void)strcat(resolved, wbuf);
}
```

https://github.com/libressl-portable/openbsd/blob/OPENBSD_2_0/src/lib/libc/stdlib/realpath.c

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.
  - 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
  - Next lecture.

Avoiding Buffer Overflows

- Train the developers to write secure code.
  - Provide developers with tools that make it easier to write secure code.
- Avoiding buffer overflow vulnerabilities requires validating the lengths of untrusted input before performing read or write operations into buffers.
- Common libc string functions do not encourage this practice and make it easy to introduce buffer overflow vulnerabilities.
- However, better alternatives are available.
- Aside: 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.

The Trouble With strc*()

```
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()` are common causes of buffer overflow vulnerabilities.
- These functions are considered unsafe across the industry.

Replacing strc*()

```
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: You have to use it right

- Vulnerability in htpasswd.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*(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.

The Trouble With strnc*()

```
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.

strl*() To The Rescue

```
size_t strlcpy(char *dst, const char *src, size_t size);
size_t strlcat(char *dst, const char *src, size_t size);
```

- In order to address the shortcomings of `strncpy()`/`strncat()`, the `strl*` family of functions were designed.
- `strlcpy()` copies up to `size-1` characters from the NULL-terminated string `src` to `dst`, NULL-terminating the result.
- `strlcat()` appends the NULL-terminated string `src` to the end of `dst`. It will append at most `size-strlen(dst)-1` bytes, NULL-terminating the result.
- The result is ALWAYS NULL-terminated.

strl*() To The Rescue

```
char buf[MAX_PATH_LEN];
/* assemble fully qualified name from provided path and file name */
if ((strlcpy(buf, path, sizeof(buf)) >= sizeof(buf))
        || (strlcat(buf, "/", sizeof(buf)) >= sizeof(buf))
        || (strlcat(buf, fname, sizeof(buf)) >= sizeof(buf)))
{ /* handle truncation error */ }
```

- The return value for both functions represents the total length of the string they tried to create, allowing the caller to detect truncation.
- Thus we can guarantee NULL-termination and prevent buffer overflows without the burden of performing complicated run time arithmetic operations.

strl*() To The Rescue

- Not everyone has embraced the salvation of strl*()

"One of the longest-running requests for the GNU C Library (glibc) is the addition of the strlcpy() family of string functions... Despite years of requests, however, the glibc maintainers have never allowed these functions to be added.

Back in 2000, one Christoph Hellwig posted a patch adding strlcpy() and strlcat() to glibc. The glibc maintainer at that time, Ulrich Drepper, rejected the patch in classic style:

> *This is horribly inefficient BSD crap. Using these function only leads to other errors. Correct string handling means that you always know how long your strings are and therefore you can you memcpy (instead of strcpy).*
>
> *Beside, those who are using strcat or variants deserved to be punished.*

...Fourteen years after Christoph's patch was posted, there is still no strlcpy() in glibc."

- Still not available in libc. Must use libbsd, or copy implementation from OpenBSD.

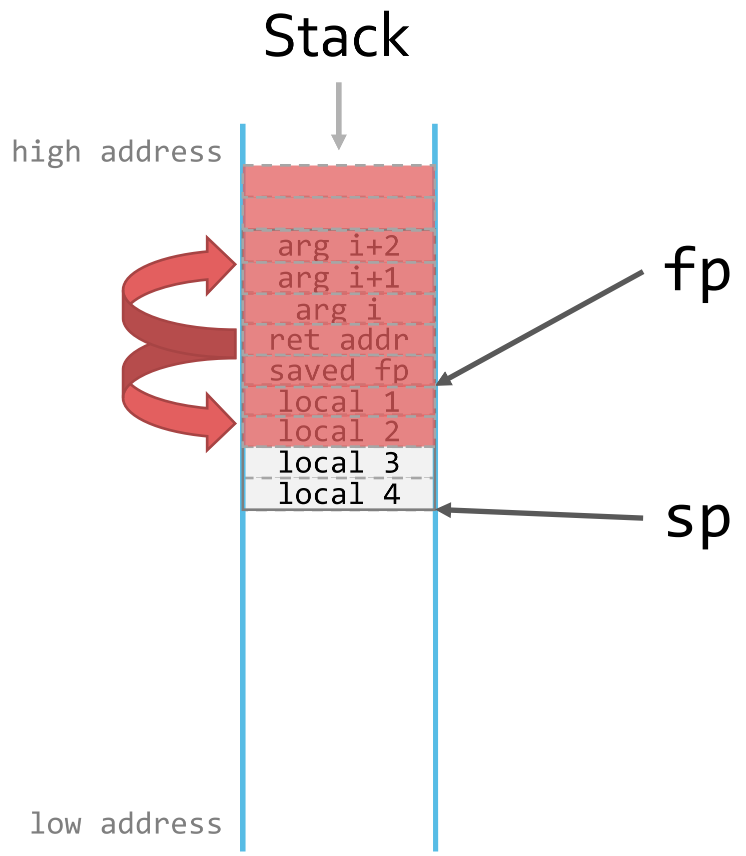
https://lwn.net/Articles/612244/

Review

- An attacker can direct the execution of your program by manipulating input data it acts on.
- Assume input can be malicious. Validate lengths and bounds before accessing arrays.
- Separate control data from user data.
- 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

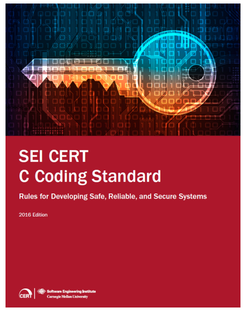


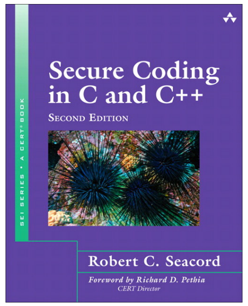
Additional Resources

- *Memory Corruption Attacks: The Almost Complete History* by Haroon Meer, Black Hat USA 2010
  - https://www.youtube.com/watch?v=stVz9rhTdQ8
- *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
- 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





Monday is holiday

Next Lecture (I think)

Low Level Software Security II:
Shellcode, Countermeasures,…