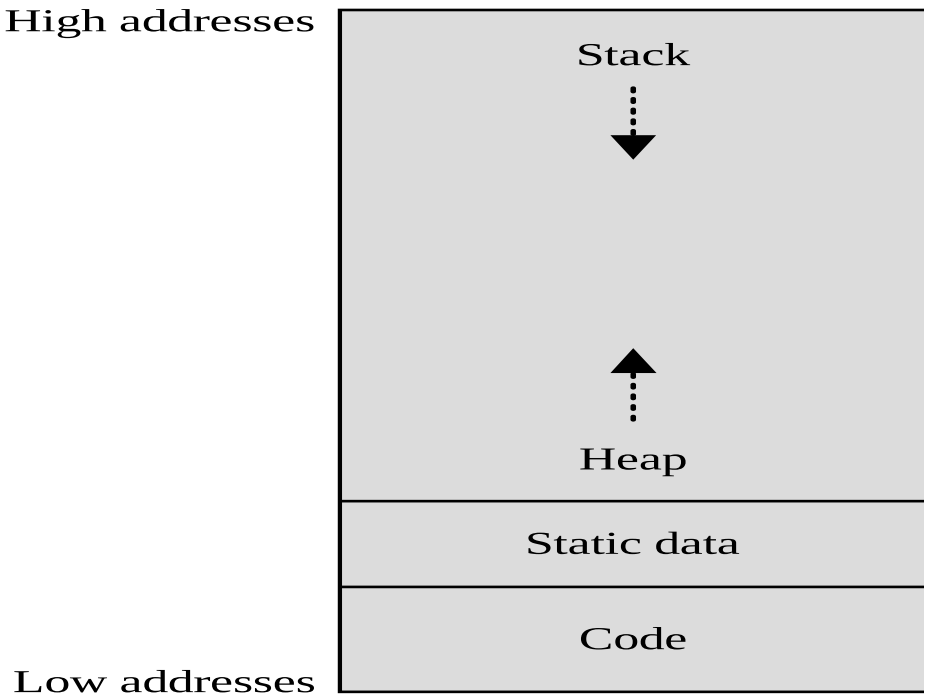
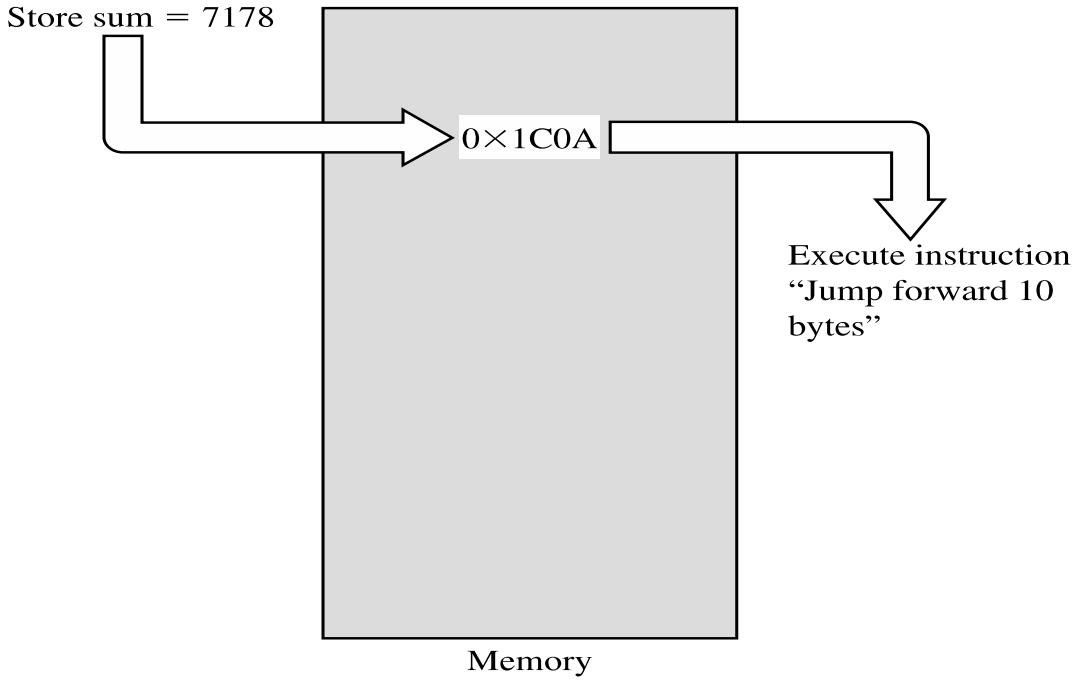
Software Security

Memory Allocation



Code and data separated, with the heap growing up toward high addresses and the stack growing down from the high addresses.

Data vs. Instructions



The same hex value in the same spot in memory can either be a meaningful data value or a meaningful instruction depending on whether the computer treats it as code or data.

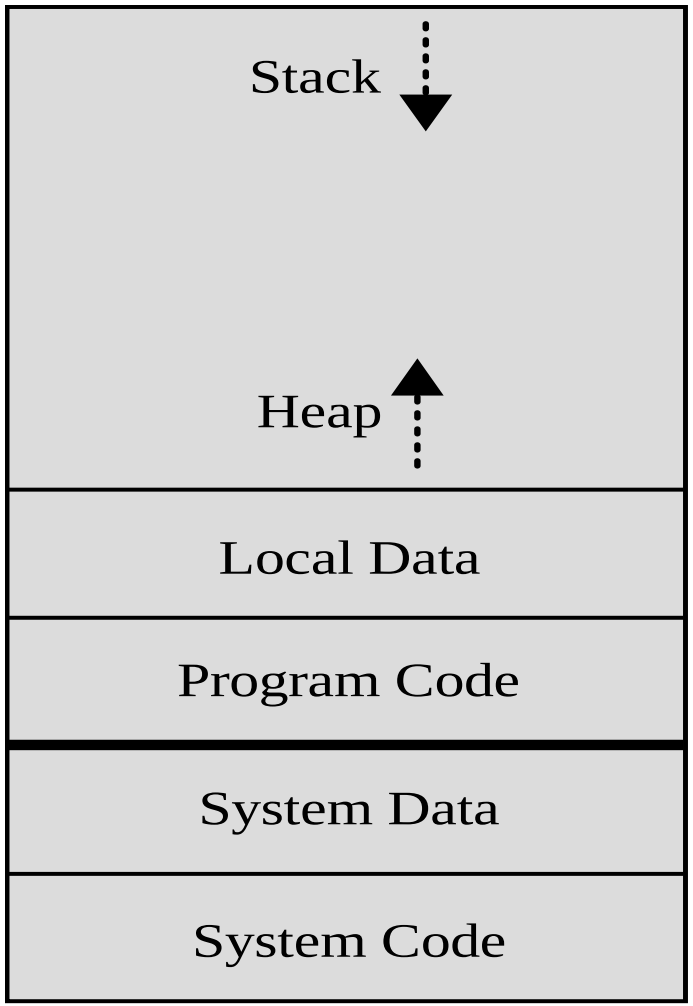
Buffer Overflows

- Occur when data is written beyond the space allocated for it, such as a 10th byte in a 9-byte array
- In a typical exploitable buffer overflow, an attacker's inputs are expected to go into regions of memory allocated for data, but those inputs are instead allowed to overwrite memory holding executable code
- The trick for an attacker is finding buffer overflow opportunities that lead to overwritten memory being executed, and finding the right code to input

```
char sample[10];

int i;

for (i=0; i<=9; i++)
  sample[i] = 'A';

sample[10] = 'B';
```

This is a very simple buffer overflow. Character B is placed in memory that wasn't allocated by or for this procedure.

Memory Organization



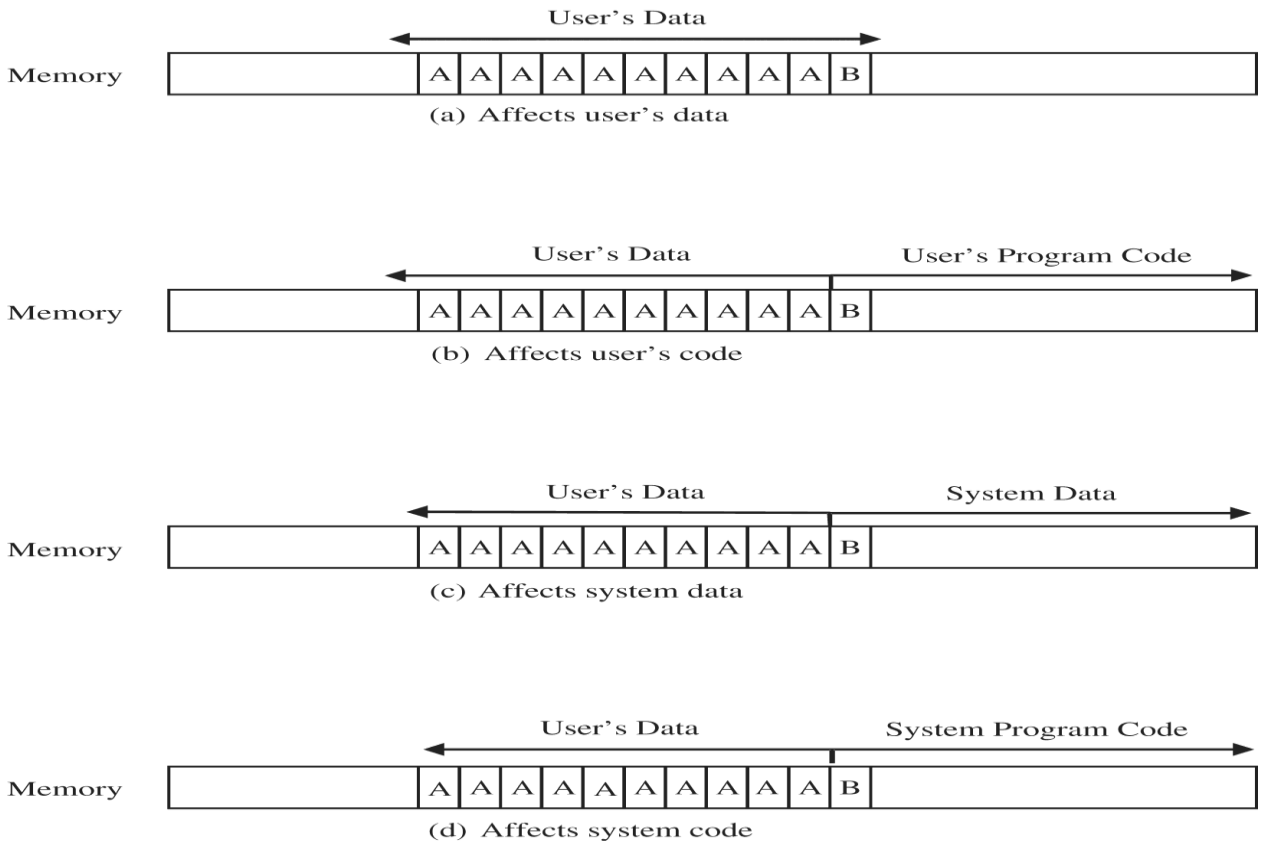
High addresses

Low addresses

We show where the system data/code reside vs. where the program code and its local data reside.

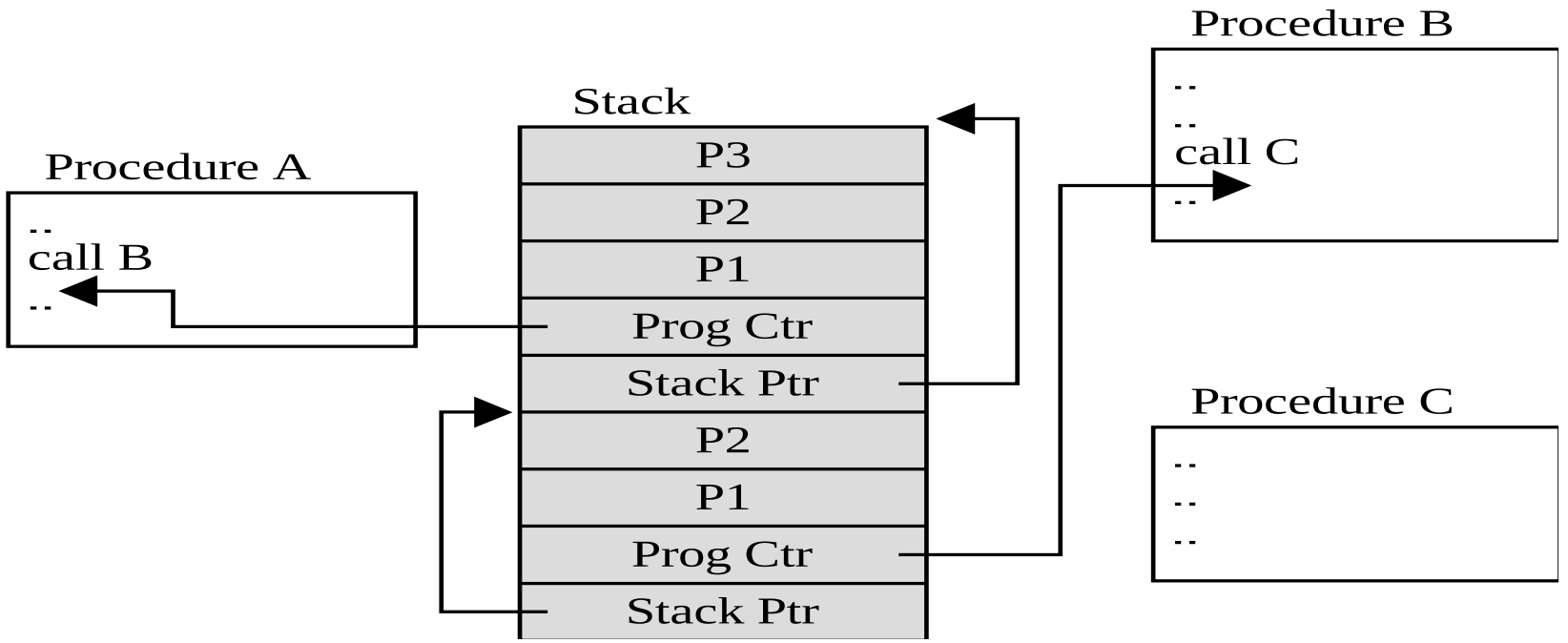
This context is important for understanding how an attack that takes place inside a given program can affect that program vs. how it can affect the rest of the system.

Where a Buffer Can Overflow



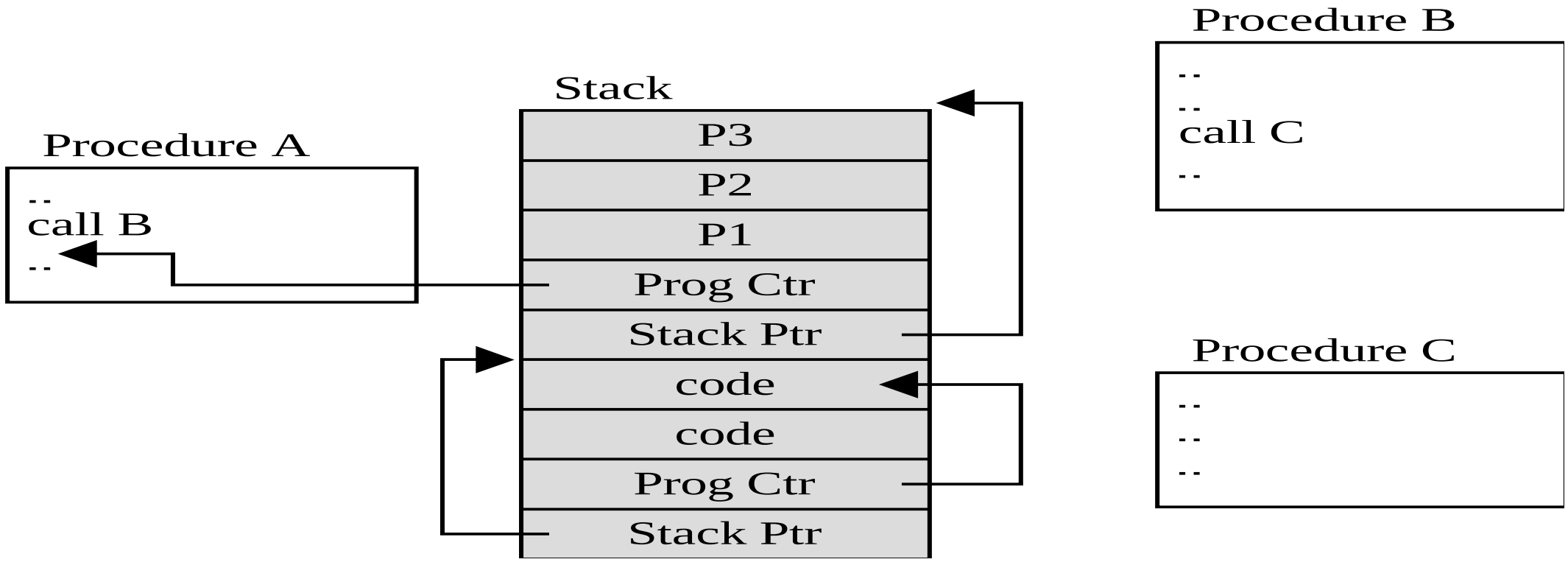
Examples of buffer overflow effects in the context of the earlier AAAAAAAAAAB example. The memory that's overwritten depends on where the buffer resides.

The Stack after Procedure Calls



When procedure A calls procedure B, procedure B gets added to the stack along with a pointer back to procedure A. In this way, when procedure B is finished running, it can get popped off the stack, and procedure A will just continue executing where it left off.

Compromised Stack



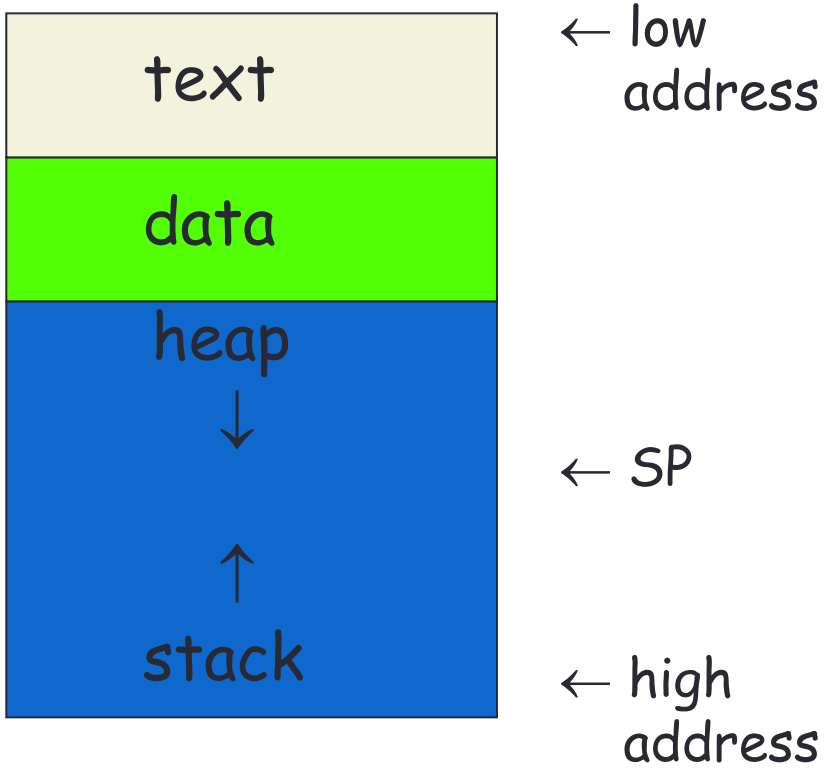
Instead of pointing at procedure B in this case, the program counter is pointing at code that's been placed on the stack as a result of an overflow.

Overwriting Memory for Execution

- Overwrite the program counter stored in the stack
- Overwrite part of the code in low memory, substituting new instructions
- Overwrite the program counter and data in the stack so that the program counter points to the stack

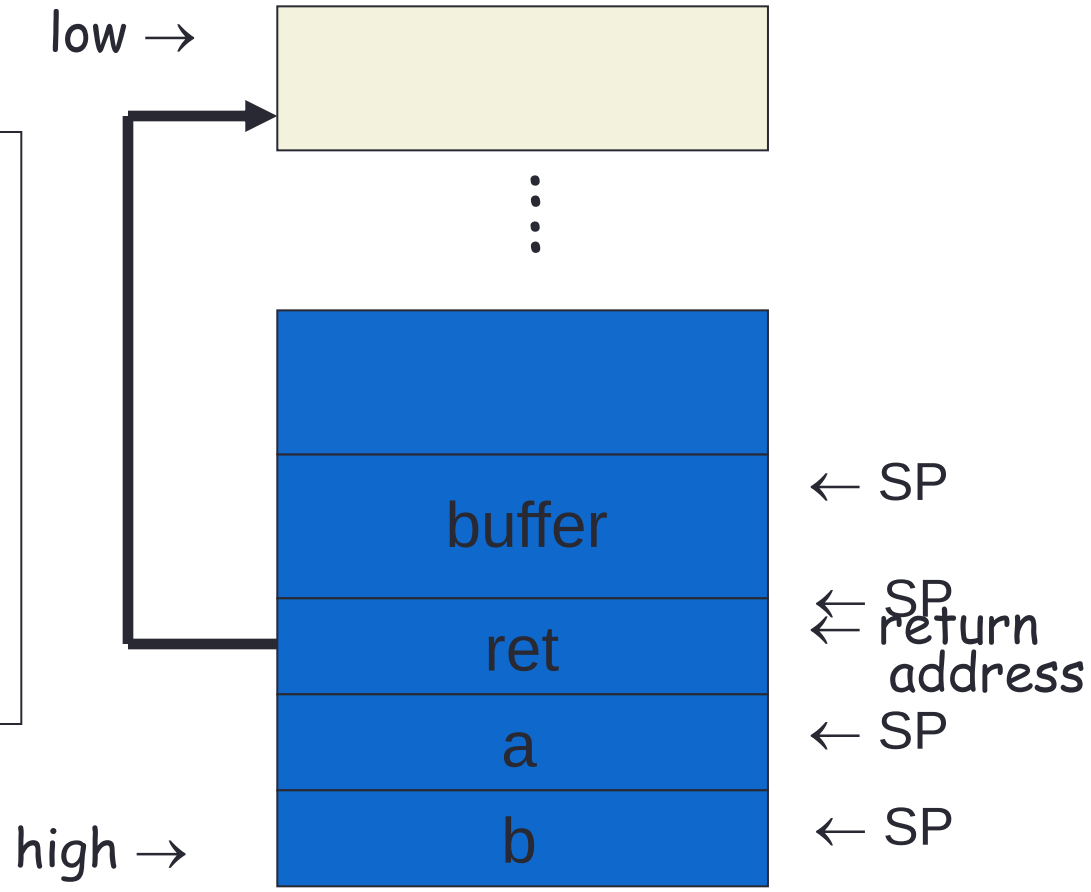
Memory Organization

- **Text** == code
- **Data** == static variables
- **Heap** == dynamic data
- **Stack** == "scratch paper"
  - Dynamic local variables
  - Parameters to functions
  - Return address



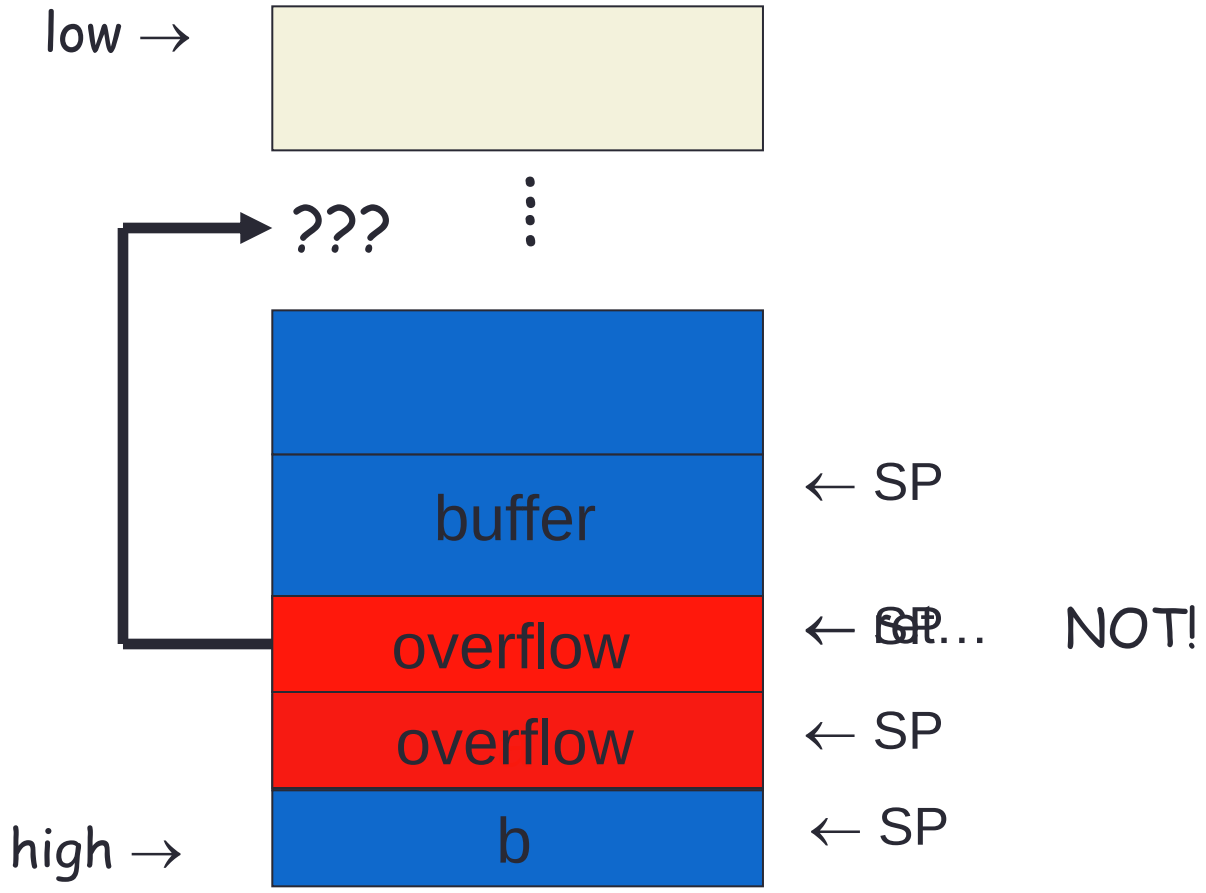
Simplified Stack Example

```
void func(int a, int b){
    char buffer[10];
}
void main(){
    func(1, 2);
}
```



Smashing the Stack

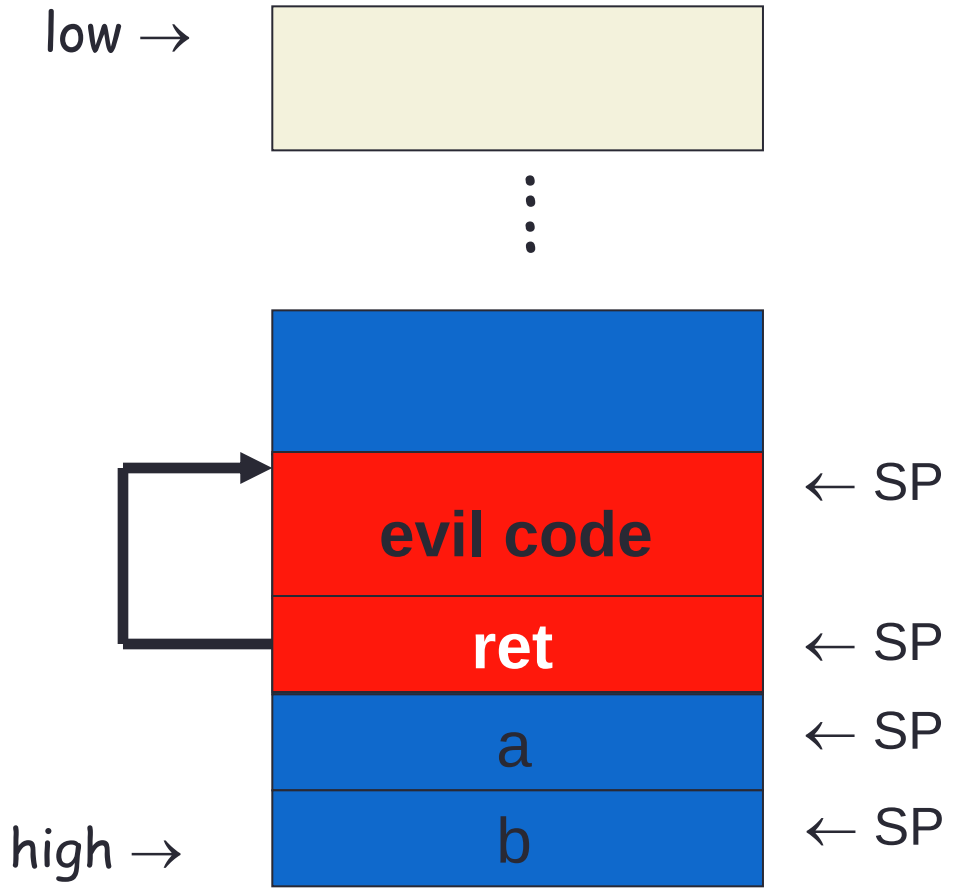
- What happens if buffer overflows?
- Program "returns" to wrong location
- A crash is likely



Mark Stamp's Information Security: Principles and Practices by Mark Stamp/ Deven Shah.

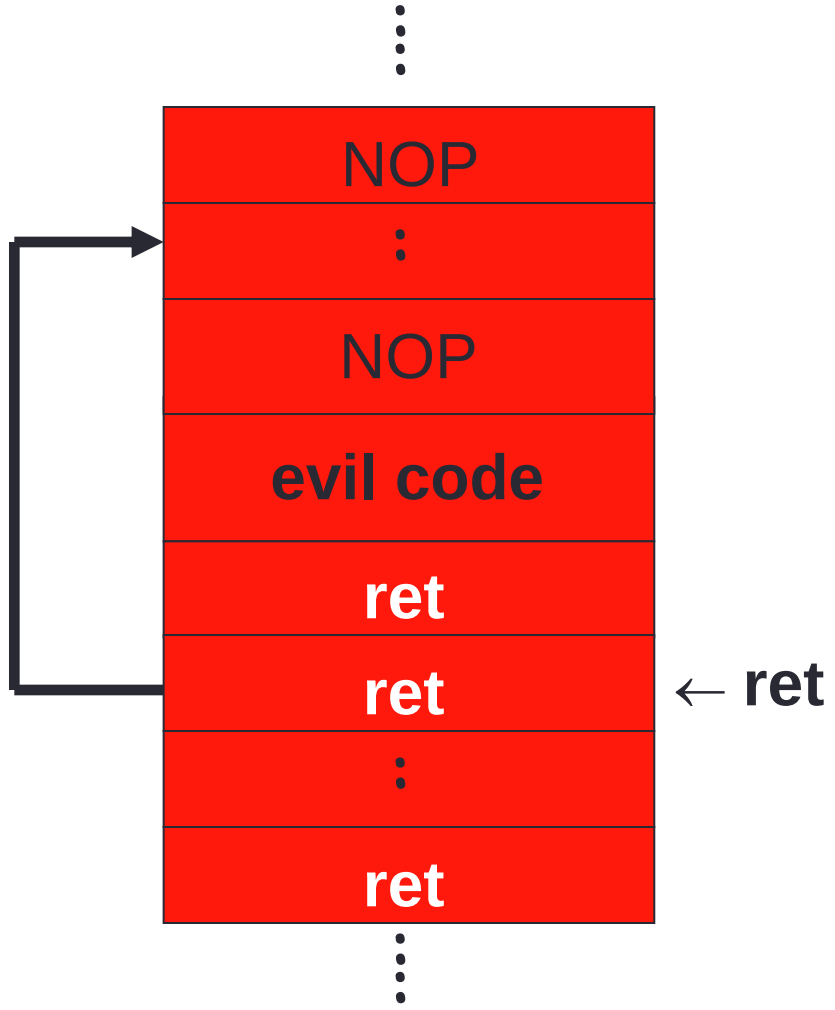
Smashing the Stack

- Trudy has a better idea…
- **Code injection**
- Trudy can run code of her choosing!



Smashing the Stack

- Trudy may not know
  - Address of evil code
  - Location of **ret** on stack
- Solutions
  - Precede evil code with NOP "landing pad"
  - Insert lots of new **ret**



Harm from Buffer Overflows

- Overwrite:
  - Another piece of your program's data
  - An instruction in your program
  - Data or code belonging to another program
  - Data or code belonging to the operating system
- Overwriting a program's instructions gives attackers that program's execution privileges
- Overwriting operating system instructions gives attackers the operating system's execution privileges

Stack Buffer Overflow - DOS

Just supplying random input which leads typically to the program crashing, demonstrates the basic stack overflow attack. And since the program has crashed, it can no longer supply the function or service it was running for.

At its simplest, then, a stack overflow can result in some form of denial-of-service attack on a system.

Of more interest to the attacker, rather than immediately crashing the program, is to have it transfer control to a location and code of the attacker's choosing.

The simplest way of doing this is for the input causing the buffer overflow to contain the desired target address at the point where it will overwrite the saved return address in the stack frame.

Then when the attacked function finishes and executes the return instruction, instead of returning to the calling function, it will jump to the supplied address instead and execute instructions from there.

Stack Buffer Overflows

Note in this program that the buffers are both the same size. This is a quite common practice in C programs.

Indeed, the standard C IO library has a defined constant BUFSIZ, which is the default size of the input buffers it uses.

The problem that may result, as it does in this example, occurs when data are being merged into a buffer that includes the contents of another buffer, such that the space needed exceeds the space available.

For the first run in 10.7b, the value read is small enough that the merged response didn't corrupt the stack frame.

For the second run, the supplied input was much too large. However, because a safe input function was used, only 15 characters were read, as shown in the following line. When this was then merged with the response string, the result was larger than the space available in the destination buffer.

It overwrote the saved frame pointer, but not the return address. So the function returned, as shown by the message printed by the main() function. But when main() tried to return, because its stack frame had been corrupted and was now some random value, the program jumped to an illegal address and crashed.

In this case the combined result was not long enough to reach the return address, but this would be possible if a larger buffer size had been used.

```
void getinp(char *inp, int siz)
{
   puts("Input value: ");
   fgets(inp, siz, stdin);
   printf("buffer3 getinp read %s\n", inp);
}

void display(char *val)
{
   char tmp[16];
   sprintf(tmp, "read val: %s\n", val);
   puts(tmp);
}

int main(int argc, char *argv[])
{
   char buf[16];
   getinp(buf, sizeof(buf));
   display(buf);
   printf("buffer3 done\n");
}
```

(a) Another stack overflow C code

```
$ cc -o buffer3 buffer3.c

$ ./buffer3
Input value:
SAFE
buffer3 getinp read SAFE
read val: SAFE
buffer3 done

$ ./buffer3
Input value:
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
buffer3 getinp read XXXXXXXXXXXXXXX
read val: XXXXXXXXXXXXXXX

buffer3 done
Segmentation fault (core dumped)
```

(b) Another stack overflow example runs

Unsafe C Standard Library Routines

This shows that when looking for buffer overflows, all possible places where externally sourced data are copied or merged have to be located.

Note that these do not even have to be in the code for a particular program, they can (and indeed do) occur in library routines used by programs, including both standard libraries and third-party application libraries.

Thus, for both attacker and defender, the scope of possible buffer overflow locations is very large.

A list of some of the most common unsafe standard C Library routines is given in Table 10.2 . These routines are all suspect and should not be used without checking the total size of data being transferred in advance, or better still by being replaced with safer alternatives.

`gets(char *str)`	read line from standard input into str
`sprintf(char *str, char *format, ...)`	create str according to supplied format and variables
`strcat(char *dest, char *src)`	append contents of string src to string dest
`strcpy(char *dest, char *src)`	copy contents of string src to string dest
`vsprintf(char *str, char *fmt, va_list ap)`	create str according to supplied format and variables

Table 10.2 : Some Common Unsafe C Standard Library Routines

Shellcode

Code supplied by attacker

- Often saved in buffer being overflowed. Traditionally transferred control to a user command-line interpreter (shell)

Machine code

- Specific to processor and operating system. Traditionally needed good assembly language skills to create

Metasploit Project

- Provides useful information to people who perform penetration and exploit research

There are several generic restrictions on the content of shellcode….it has to be **position independent . That means it cannot contain any absolute address** referring to itself, because the attacker generally cannot determine in advance exactly where the targeted buffer will be located in the stack frame of the function in which it is defined.

This means shellcode is specific to a processor architecture, and indeed usually to an OS, as it needs to run on targeted system and interact with its system functions.

This is the major reason why buffer overflow attacks are usually targeted at a specific piece of software running on a specific operating system.

Because shellcode is machine code, writing it traditionally required a good understanding of the assembly language and operation of the targeted system.

However, more recently a number of sites and tools have been developed that automate this process thus making the development of shellcode exploits available to a much larger potential audience

```
int main(int argc, char *argv[])
{
   char *sh;
   char *args[2];

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

(a) Desired shellcode code in C

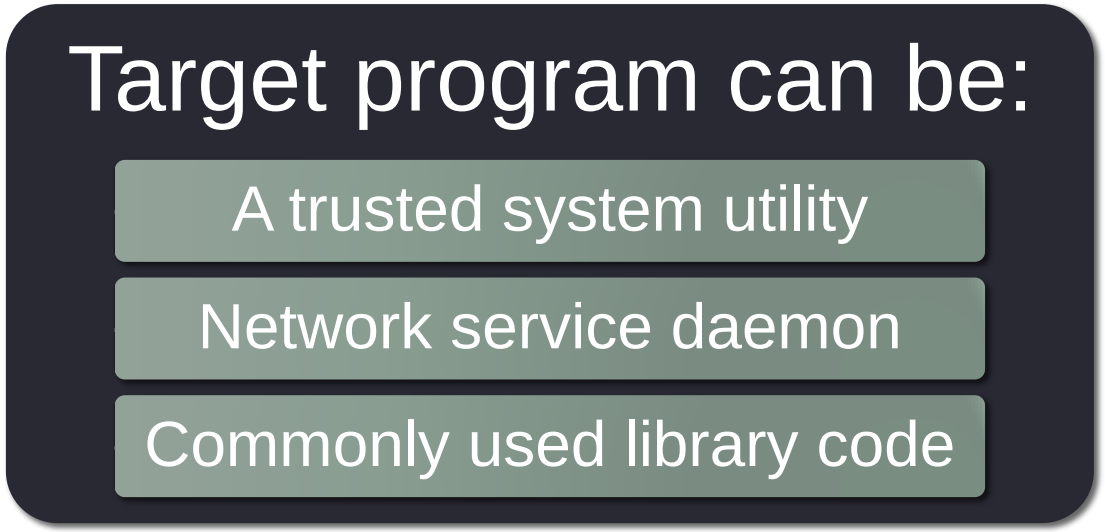
```
      nop
      nop                 // end of nop sled
      jmp   find          // jump to end of code
cont: pop   %esi          // pop address of sh off stack into %esi
      xor   %eax,%eax     // zero contents of EAX
      mov   %al,0x7(%esi) // copy zero byte to end of string sh (%esi)
      lea   (%esi),%ebx   // load address of sh (%esi) into %ebx
      mov   %ebx,0x8(%esi) // save address of sh in args[0] (%esi+8)
      mov   %eax,0xc(%esi) // copy zero to args[1] (%esi+c)
      mov   $0xb,%al      // copy execve syscall number (11) to AL
      mov   %esi,%ebx     // copy address of sh (%esi) t0 %ebx
      lea   0x8(%esi),%ecx // copy address of args (%esi+8) to %ecx
      lea   0xc(%esi),%edx // copy address of args[1] (%esi+c) to %edx
      int   $0x80         // software interrupt to execute syscall
find: call  cont          // call cont which saves next address on stack
sh:   .string "/bin/sh "  // string constant
args: .long 0             // space used for args array
      .long 0             // args[1] and also NULL for env array
```

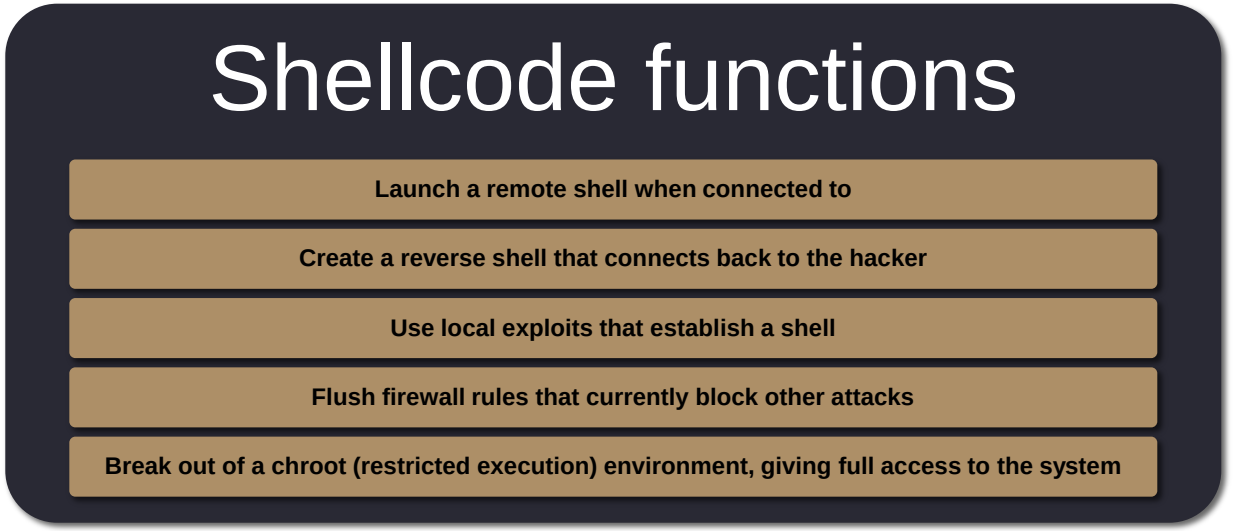
(b) Equivalent position-independent x86 assembly code

```
90 90 eb 1a 5e 31 c0 88 46 07 8d 1e 89 5e 08 89
46 0c b0 0b 89 f3 8d 4e 08 8d 56 0c cd 80 e8 e1
ff ff ff 2f 62 69 6e 2f 73 68 20 20 20 20 20 20
```

(c) Hexadecimal values for compiled x86 machine code

Stack Overflow Variants





The targeted program need not be a trusted system utility. Another possible target is a program providing a network service; that is, a network daemon. A common approach for such programs is listening for connection requests from clients and then spawning a child process to handle that request. The child process typically has the network connection mapped to its standard input and output. This means the child program's code may use the same type of unsafe input or buffer copy code as we've seen already.

This was indeed the case with the stack overflow attack used by the Morris Worm back in 1988. It targeted the use of gets() in the fingerd daemon handling requests for the UNIX finger network service (which provided information on the users on the system).

Yet another possible target is a program, or library code, which handles common document formats (e.g., the library routines used to decode and display GIF or JPEG images). In this case, the input is not from a terminal or network connection, but from the file being decoded and displayed.

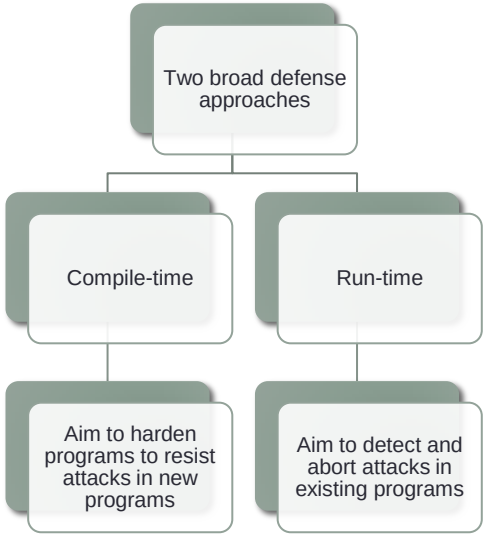
If such code contains a buffer overflow, it can be triggered as the file contents are read, with the details encoded in a specially corrupted image. This attack file would be distributed via e-mail, instant messaging, or as part of a Web page. Because the attacker is not directly interacting with the targeted program and system, the shellcode would typically open a network connection back to a system under the attacker's control, to return information and possibly receive additional commands to execute.

All of this shows that buffer overflows can be found in a wide variety of programs, processing a range of different input, and with a variety of possible responses.

30 cybersecurity search tools

1. Dehashed—View leaked credentials.
2. SecurityTrails—Extensive DNS data.
3. DorkSearch—Really fast Google dorking.
4. ExploitDB—Archive of various exploits.
5. ZoomEye—Gather information about targets.
6. Pulsedive—Search for threat intelligence.
7. GrayHatWarfare—Search public S3 buckets.
8. PolySwarm—Scan files and URLs for threats.
9. Fofa—Search for various threat intelligence.
10. LeakIX—Search publicly indexed information.
11. DNSDumpster—Search for DNS records quickly.
12. FullHunt—Search and discovery attack surfaces.
13. AlienVault—Extensive threat intelligence feed.
14. ONYPHE—Collects cyber-threat intelligence data.
15. Grep App—Search across a half million git repos.
16. URL Scan—Free service to scan and analyse websites.
17. Vulners—Search vulnerabilities in a large database.
18. WayBackMachine—View content from deleted websites.
19. Shodan—Search for devices connected to the internet.
20. Netlas—Search and monitor internet connected assets.
21. CRT sh—Search for certs that have been logged by CT.
22. Wigle—Database of wireless networks, with statistics.
23. PublicWWW—Marketing and affiliate marketing research.
24. Binary Edge—Scans the internet for threat intelligence.
25. GreyNoise—Search for devices connected to the internet.
26. Hunter—Search for email addresses belonging to a website.
27. Censys—Assessing attack surface for connected devices.
28. IntelligenceX—Search Tor, I2P, data leaks, domains, and emails.
29. Packet Storm Security—Browse latest vulnerabilities and exploits.
30. SearchCode—Search 75 billion lines of code from 40m projects.

Buffer Overflow Defenses



There is a need to defend systems against such attacks by either preventing them, or at least detecting and aborting such attacks. These can be broadly classified into two categories:

• Compile-time defenses, which aim to harden programs to resist attacks in new programs

• Run-time defenses, which aim to detect and abort attacks in existing programs

While suitable defenses have been known for a couple of decades, the very large existing base of vulnerable software and systems hinders their deployment. Hence the interest in run-time defenses, which can be deployed as operating systems and updates and can provide some protection for existing vulnerable programs.

Compile-time defenses aim to prevent or detect buffer overflows by instrumenting programs when they are compiled. The possibilities for doing this range from choosing a high-level language that does not permit buffer overflows, to encouraging safe coding standards, using safe standard libraries, or including additional code to detect corruption of the stack frame.

Compile-Time Defenses: Programming Language

- Use a modern high-level language
  - Not vulnerable to buffer overflow attacks
  - Compiler enforces range checks and permissible operations on variables

Disadvantages

- Additional code must be executed at run time to impose checks
- Flexibility and safety comes at a cost in resource use
- Distance from the underlying machine language and architecture means that access to some instructions and hardware resources is lost
- Limits their usefulness in writing code, such as device drivers, that must interact with such resources

Compile-Time Defenses: Safe\ Coding Techniques

- C designers placed much more emphasis on space efficiency and performance considerations than on type safety
  - Assumed programmers would exercise due care in writing code
- Programmers need to inspect the code and rewrite any unsafe coding
  - An example of this is the OpenBSD project
- Programmers have audited the existing code base, including the operating system, standard libraries, and common utilities
  - This has resulted in what is widely regarded as one of the safest operating systems in widespread use

Compile-Time Defenses: Language Extensions/Safe Libraries

- Handling dynamically allocated memory is more problematic because the size information is not available at compile time
  - Requires an extension and the use of library routines
    - Programs and libraries need to be recompiled
    - Likely to have problems with third-party applications
- Concern with C is use of unsafe standard library routines
  - One approach has been to replace these with safer variants
    - Libsafe is an example
    - Library is implemented as a dynamic library arranged to load before the existing standard libraries

Compile-Time Defenses: Stack Protection

- Add function entry and exit code to check stack for signs of corruption
- Use random canary
  - Value needs to be unpredictable
  - Should be different on different systems
- Stackshield and Return Address Defender (RAD)
  - GCC extensions that include additional function entry and exit code
    - Function entry writes a copy of the return address to a safe region of memory
    - Function exit code checks the return address in the stack frame against the saved copy
    - If change is found, aborts the program

Run-Time Defenses: Executable Address Space Protection

Use virtual memory support to make some regions of memory non-executable

- Requires support from memory management unit (MMU)
- Long existed on SPARC / Solaris systems
- Recent on x86 Linux/Unix/Windows systems

Issues

- Support for executable stack code
- Special provisions are needed

Run-Time Defenses: Address Space Randomization

- Manipulate location of key data structures
  - Stack, heap, global data - Using random shift for each process
  - Large address range on modern systems means wasting some has negligible impact
- Randomize location of heap buffers
- Random location of standard library functions

Run-Time Defenses: Guard Pages

- Place guard pages between critical regions of memory
  - Flagged in MMU as illegal addresses
  - Any attempted access aborts process
- Further extension places guard pages Between stack frames and heap buffers
  - Cost in execution time to support the large number of page mappings necessary

Input Size & Buffer Overflow

- Programmers often make assumptions about the maximum expected size of input
  - Allocated buffer size is not confirmed
  - Resulting in buffer overflow
- Testing may not identify vulnerability
  - Test inputs are unlikely to include large enough inputs to trigger the overflow
- Safe coding treats all input as dangerous

Interpretation of Program Input

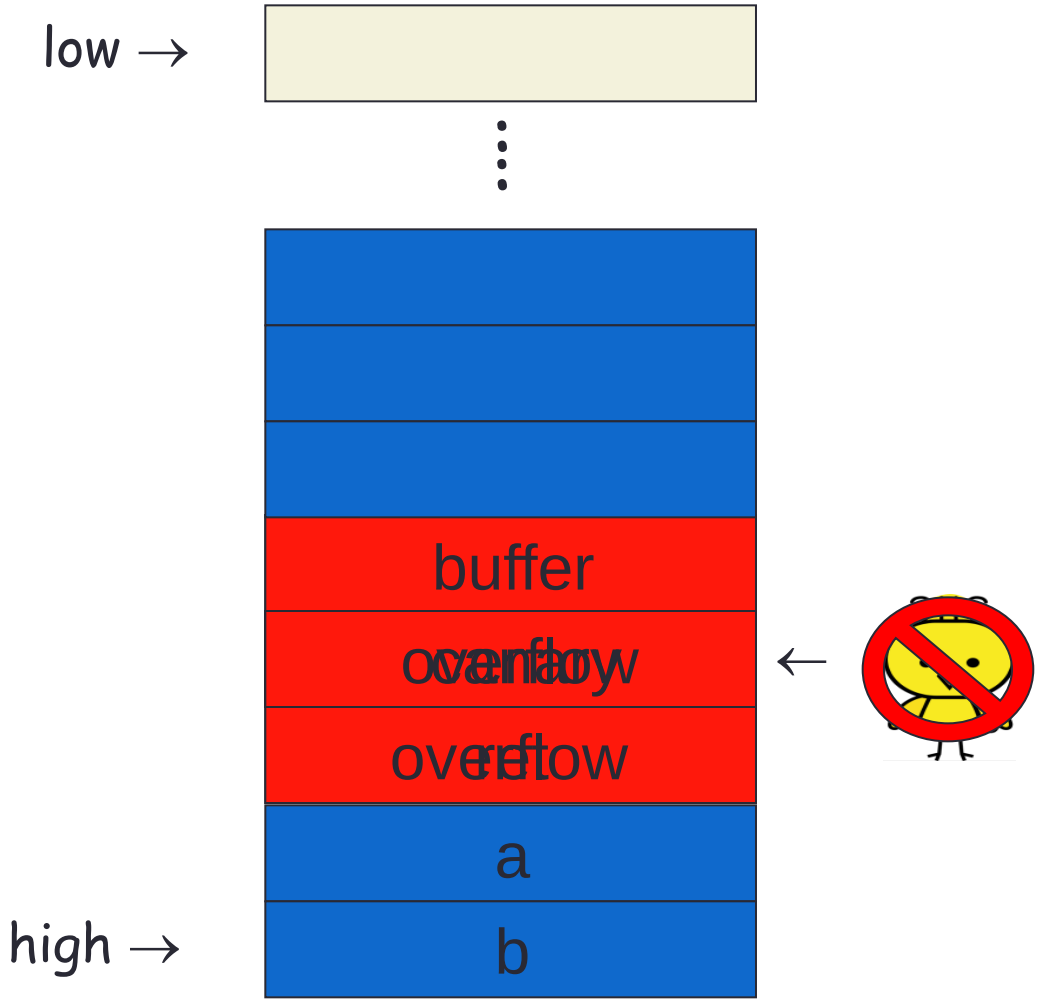
- Program input may be binary or text
  - Binary interpretation depends on encoding and is usually application specific
- There is an increasing variety of character sets being used
  - Care is needed to identify just which set is being used and what characters are being read
- Failure to validate may result in an exploitable vulnerability
- 2014 Heartbleed OpenSSL bug is a recent example of a failure to check the validity of a binary input value…….next

Stack Smashing Prevention

- 1st choice: employ **non-executable stack**
  - "No execute" **NX bit** (if available)
  - Seems like the logical thing to do, but some real code executes on the stack (Java does this)
- 2nd choice: use **safe languages** (Java, C#)
- 3rd choice: use **safer C functions**
  - For unsafe functions, there are safer versions
  - For example, strncpy instead of strcpy

Stack Smashing Prevention

- **Canary**
  - Run-time stack check
  - Push canary onto stack
  - Canary value:
    - Constant 0x000aff0d
    - Or value depends on **ret**



Mark Stamp's Information Security: Principles and Practices by Mark Stamp/ Deven Shah.

Microsoft's Canary

- Microsoft added **buffer security check** feature to C++ with /GS compiler flag
- Uses canary (or "security cookie")
- **Q:** What to do when canary dies?
- **A:** Check for user-supplied handler
- Handler may be subject to attack
  - Claimed that attacker can specify handler code
  - If so, "safe" buffer overflows become exploitable when /GS is used!

Buffer Overflow

- The “attack of the decade” for 90’s
- Will be the attack of the decade for 00’s
- Can be prevented
  - Use safe languages/safe functions
  - Educate developers, use tools, etc.
- Buffer overflows will exist for a long time
  - Legacy code
  - Bad software development

Overflow Countermeasures Summary

- Staying within bounds
  - Check lengths before writing
  - Confirm that array subscripts are within limits
  - Double-check boundary condition code for off-by-one errors
  - Limit input to the number of acceptable characters
  - Limit programs' privileges to reduce potential harm
- Many languages have overflow protections
- Code analyzers can identify many overflow vulnerabilities
- Canary values in stack to signal modification

Incomplete Mediation

Input Validation

- Consider: strcpy(buffer, argv[1])
- A buffer overflow occurs if

  len(buffer) < len(argv[1])
- Software must **validate** the input by checking the length of argv[1]
- Failure to do so is an example of a more general problem: **incomplete mediation**

Input Validation

- Consider web form data
- Suppose input is validated on client
- For example, the following is valid

  http://www.things.com/orders/final&custID=112&num=55A&qty=20&price=10&shipping=5&total=205

- Suppose input is not checked on server
  - Why bother since input checked on client?
  - Then attacker could send http message

  http://www.things.com/orders/final&custID=112&num=55A&qty=20&price=10&shipping=5&total=25

Incomplete Mediation

- Linux kernel
  - Research has revealed many buffer overflows
  - Many of these are due to incomplete mediation
- Linux kernel is “good” software since
  - Open-source
  - Kernel — written by coding gurus
- Tools exist to help find such problems
  - But incomplete mediation errors can be subtle
  - And tools useful to attackers too!

Incomplete Mediation

- **Validate All Input**
- **Guard Against Users' Fingers**
  - There is no reason to leave sensitive data under control of an untrusted user.

Time-of-Check to Time-of-Use

- The time-of-check totime-of-use (TOCTTOU) flaw concerns mediation that is performed with a “bait and switch” in the middle.
- Between access check and use, data must be protected against change.

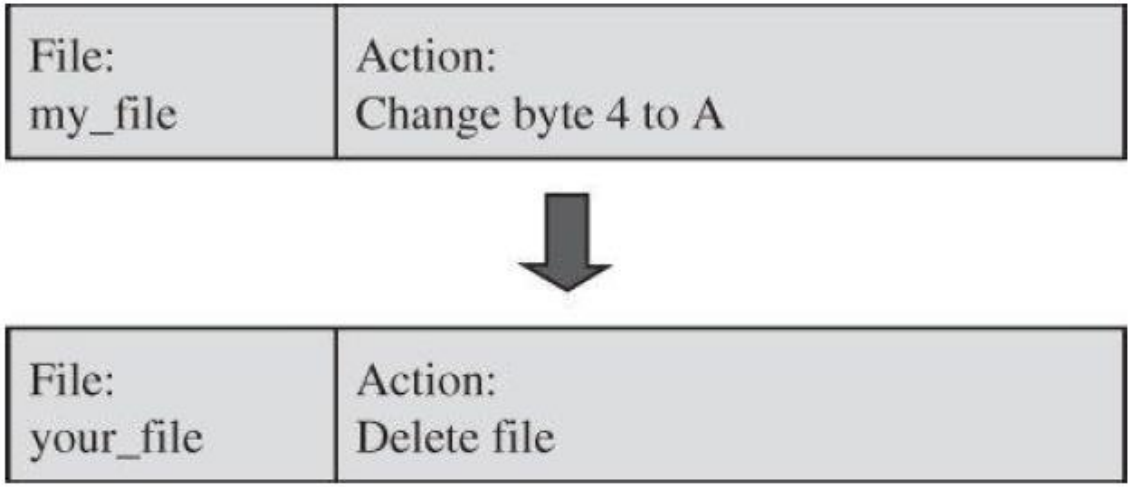


FIGURE 3-13 Unchecked Change to Work Descriptor

Time-of-Check to Time-of-Use

- The problem is called a time-of-check to time-of-use flaw because it exploits the delay between the two actions: check and use.
- That is, between the time the access was checked and the time the result of the check was used, a change occurred, invalidating the result of the check.

Security Implication

- The security implication here is clear: Checking one action and performing another is an example of ineffective access control, leading to confidentiality failure or integrity failure or both.
- We must be wary whenever a time lag or loss of control occurs, making sure that there is no way to corrupt the check's results during that interval.

Time-of-Check to Time-of-Use : Countermeasures

- There are ways to prevent exploitation of the time lag, again depending on our security tool, access control.
- Critical parameters are not exposed during any loss of control. The access-checking software must own the request data until the requested action is complete.
- Another protection technique is to ensure serial integrity, that is, to allow no interruption (loss of control) during the validation.
- Or the validation routine can initially copy data from the user's space to the routine's area—out of the user's reach—and perform validation checks on the copy.
- Finally, the validation routine can seal the request data to detect modification.
- Really, all these protection methods are expansions on the tamperproof criterion for a reference monitor: Data on which the access control decision is based and the result of the decision must be outside the domain of the program whose access is being controlled.

Undocumented Access Point

- During program development and testing, the programmer needs a way to access the internals of a module.
- The programmer creates an undocumented entry point or execution mode.
- Sometimes, however, the programmer forgets to remove these entry points when the program moves from development to product.
- Or the programmer decides to leave them in to facilitate program maintenance later; the programmer may believe that nobody will find the special entry.
- Programmers can be naïve, because if there is a hole, someone is likely to find it.

Undocumented Access Point

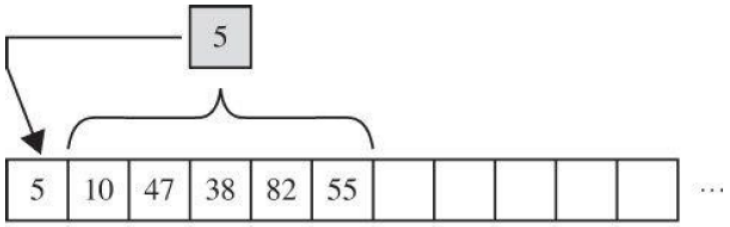
- Backdoor : An undocumented access point is called a backdoor or trapdoor.
- Such an entry can transfer control to any point with any privileges the programmer wanted.
- Few things remain secret on the web for long; someone finds an opening and exploits it.
- Thus, coding a supposedly secret entry point is an opening for unannounced visitors.

Undocumented Access Point - Protecting Against Unauthorized Entry

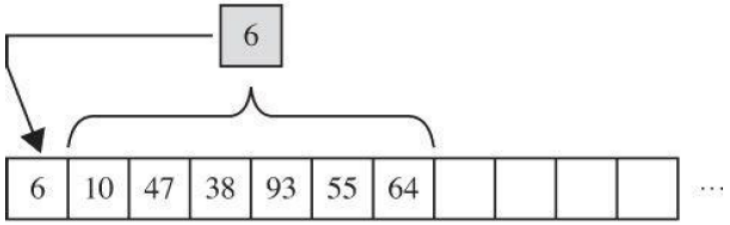
- Undocumented entry points are a poor programming practice (but they will still be used).
- They should be found during rigorous code reviews in a software development process.
- But preventing or locking these vulnerable doorways is difficult, especially because the people who write them may not appreciate their security implications.

Off-by-One Error

- When learning to program, neophytes (novice user) can easily fail with the off-by-one error:
- miscalculating the condition to end a loop (repeat while i< = n or i<n? repeat until i=n or i>n ?)
- or overlooking that an array of A[0] through A[n] contains n+1 elements.



(a) First open issues list



(b) Second open issues list

Off-by-One Error

- A similar problem occurs when a procedure edits or reformats input, perhaps changing a one-character sequence into two or more characters (as for example,
- when the one-character ellipsis symbol “…” available in some fonts is converted by a word processor into three successive periods to account for more limited fonts.)
- These unanticipated changes in size can cause changed data to no longer fit in the space where it was originally stored.
- Worse, the error will appear to be sporadic, occurring only when the amount of data exceeds the size of the allocated space.
- Alas, the only control against these errors is correct programming: always checking to ensure that a container is large enough for the amount of data it is to contain.

Integer Overflow

- An integer overflow is a peculiar type of overflow, in that its outcome is somewhat different from that of the other types of overflows.
- An **integer overflow** occurs because a storage location is of fixed, finite size and therefore can contain only integers up to a certain limit.
- The overflow depends on whether the data values are signed (that is, whether one bit is reserved for indicating whether the number is positive or negative).

Integer Overflow

- When a computation causes a value to exceed one of the limits in Table 3-1, the extra data does not spill over to affect adjacent data items.
- That's because the arithmetic is performed in a hardware register of the processor, not in memory.

Word Size	Signed Values	Unsigned Values
8 bits	−128 to +127	0 to 255 ($2^8 - 1$)
16 bits	−32,768 to +32,767	0 to 65,535 ($2^{16} - 1$)
32 bits	−2,147,483,648 to +2,147,483,647	0 to 4,294,967,296 ($2^{32} - 1$)

TABLE 3-1 Value Range by Word Size

Integer Overflow

- Instead, either a hardware program exception or fault condition is signaled, which causes transfer to an error handling routine, or the excess digits on the most significant end of the data item are lost.
- Checking for this type of overflow is difficult, because only when a result overflows can the program determine an overflow occurs.
- Using 8-bit unsigned values, for example, a program could determine that the first operand was 147 and then check whether the second was greater than 108.
- Such a test requires double work: First determine the maximum second operand that will be in range and then compute the sum.
- Some compilers generate code to test for an integer overflow and raise an exception.

Unterminated Null-Terminated String

- Long strings are the source of many buffer overflows.
- Sometimes an attacker intentionally feeds an overly long string into a processing program to see if and how the program will fail.
- Other times the vulnerability has an accidental cause:
- A program mistakenly overwrites part of a string, causing the string to be interpreted as longer than it really is.
- How these errors actually occur depends on how the strings are stored, which is a function of the programming language, application program, and operating system involved.

Unterminated Null-Terminated String

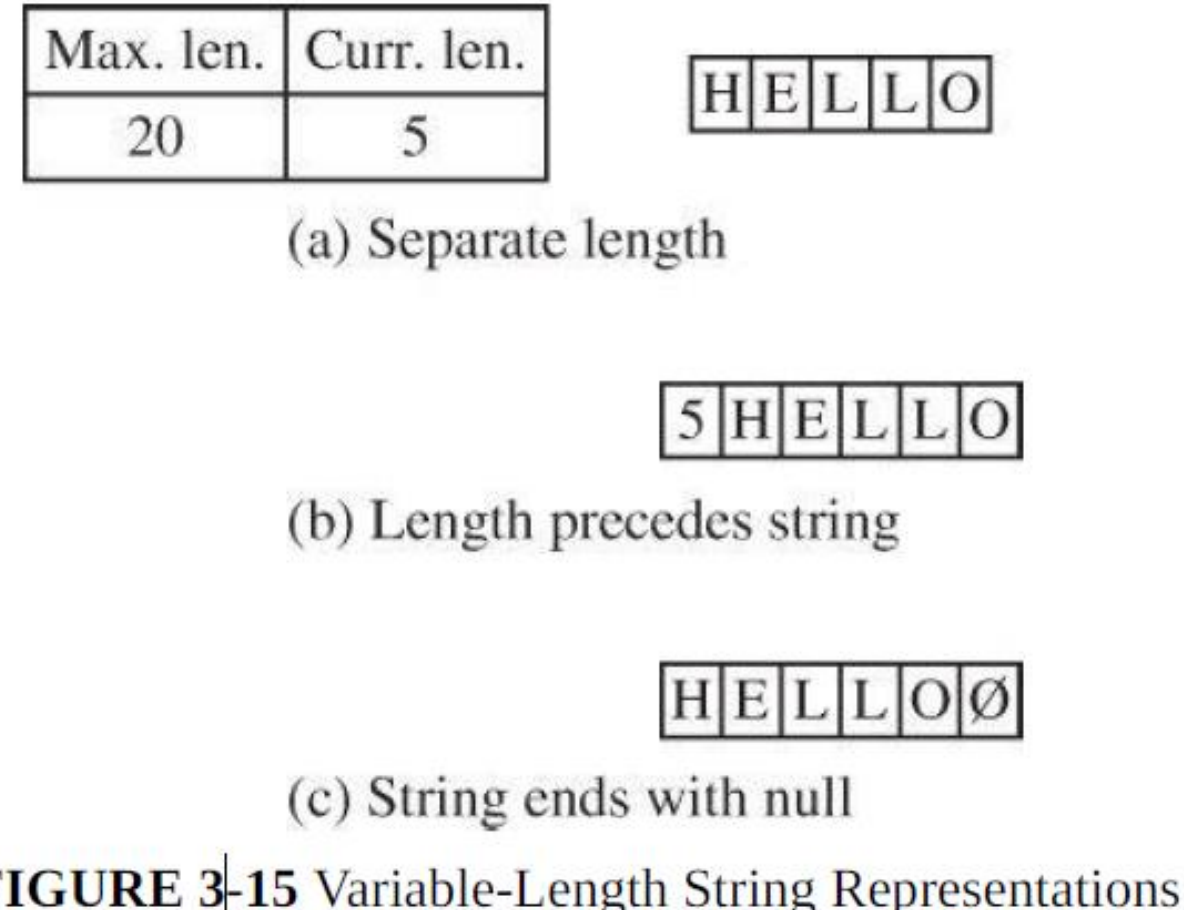


FIGURE 3-15 Variable-Length String Representations

Unterminated Null-Terminated String

- The last mode of representing a string, typically used in C, is called null terminated, meaning that the end of the string is denoted by a null byte, or 0x00
- This format is prone to misinterpretation.
- Suppose an erroneous process happens to overwrite the end of the string and its terminating null character;
- in that case, the application reading the string will continue reading memory until a null byte happens to appear (from some other data value), at any distance beyond the end of the string.
- Thus, the application can read 1, 100 to 100,000 extra bytes or more until it encounters a null.
- The problem of buffer overflow arises in computation, as well.
- Functions to move and copy a string may cause overflows in the stack or heap as parameters are passed to these functions.

Parameter Length, Type, and Number

- Another source of data-length errors is procedure parameters, from web or conventional applications.
- Among the sources of problems are these:
  - *Too many parameters*
  - *Wrong output type or size*
  - *Too-long string*

Unsafe Utility Program

- Programming languages, especially C, provide a library of utility routines to assist with common activities, such as moving and copying strings.
- In C the function strcpy (dest, src) copies a string from src to dest, stopping on a null, with the potential to overrun allocated memory.
- A safer function is strncpy (dest, src, max), which copies up to the null delimiter or max characters, whichever comes first.
- Although there are other sources of overflow problems, from these descriptions you can readily see why so many problems with buffer overflows occur.

Race Conditions

Race Condition

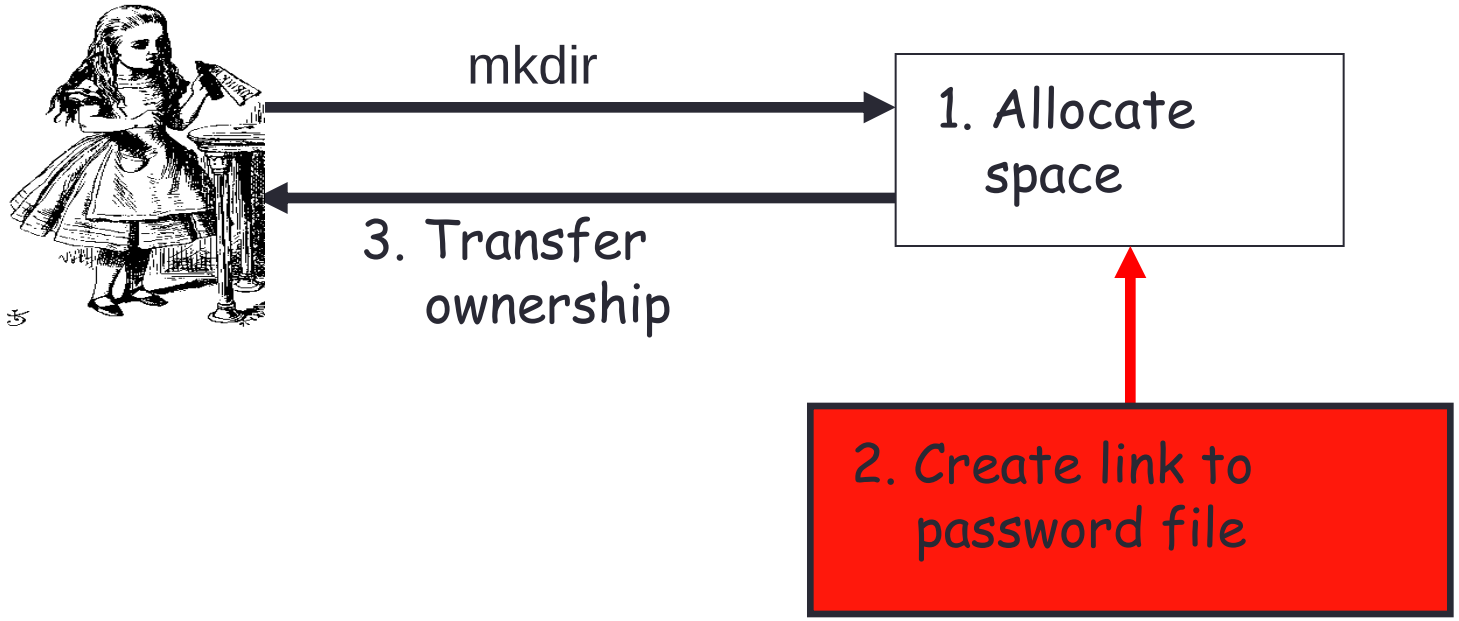
- Security processes should be **atomic**
  - Occur “all at once”
- Race conditions can arise when security-critical process occurs in stages
- Attacker makes change between stages
  - Often, between stage that gives authorization, but before stage that transfers ownership
- Example: Unix mkdir

mkdir Race Condition

- mkdir creates new directory
- How mkdir is supposed to work

mkdir Attack

- The mkdir **race condition**



- Not really a "race"
  - But attacker's timing is critical

Race Conditions

- Race conditions are common
- Race conditions may be more prevalent than buffer overflows
- But race conditions harder to exploit
  - Buffer overflow is “low hanging fruit” today
- To prevent race conditions, make security-critical processes atomic
  - Occur all at once, not in stages
  - Not always easy to accomplish in practice

Malware

- Programs planted by an agent with malicious intent to cause unanticipated or undesired effects
- Virus
  - A program that can replicate itself and pass on malicious code to other nonmalicious programs by modifying them
- Worm
  - A program that spreads copies of itself through a network
- Trojan horse
  - Code that, in addition to its stated effect, has a second, nonobvious, malicious effect

Types of Malware

Code Type	Characteristics
Virus	Code that causes malicious behavior and propagates copies of itself to other programs
Trojan horse	Code that contains unexpected, undocumented, additional functionality
Worm	Code that propagates copies of itself through a network; impact is usually degraded performance
Rabbit	Code that replicates itself without limit to exhaust resources
Logic bomb	Code that triggers action when a predetermined condition occurs
Time bomb	Code that triggers action when a predetermined time occurs
Dropper	Transfer agent code only to drop other malicious code, such as virus or Trojan horse
Hostile mobile code agent	Code communicated semi-autonomously by programs transmitted through the web
Script attack, JavaScript, Active code attack	Malicious code communicated in JavaScript, ActiveX, or another scripting language, downloaded as part of displaying a web page

Code Type	Characteristics
RAT (remote access Trojan)	Trojan horse that, once planted, gives access from remote location
Spyware	Program that intercepts and covertly communicates data on the user or the user's activity
Bot	Semi-autonomous agent, under control of a (usually remote) controller or "herder"; not necessarily malicious
Zombie	Code or entire computer under control of a (usually remote) program
Browser hijacker	Code that changes browser settings, disallows access to certain sites, or redirects browser to others
Rootkit	Code installed in "root" or most privileged section of operating system; hard to detect
Trapdoor or backdoor	Code feature that allows unauthorized access to a machine or program; bypasses normal access control and authentication
Tool or toolkit	Program containing a set of tests for vulnerabilities; not dangerous itself, but each successful test identifies a vulnerable host that can be attacked
Scareware	Not code; false warning of malicious code attack

Harm from Malicious Code

- Harm to users and systems:
  - Sending email to user contacts
  - Deleting or encrypting files
  - Modifying system information, such as the Windows registry
  - Stealing sensitive information, such as passwords
  - Attaching to critical system files
  - Hide copies of malware in multiple complementary locations
- Harm to the world:
  - Some malware has been known to infect millions of systems, growing at a geometric rate
  - Infected systems often become staging areas for new infections

Transmission and Propagation

- Setup and installer program
- Attached file
- Document viruses
- Autorun
- Using nonmalicious programs:
  - Appended viruses
  - Viruses that surround a program

Malware Activation

- One-time execution (implanting)
- Boot sector viruses
- Memory-resident viruses
- Application files
- Code libraries

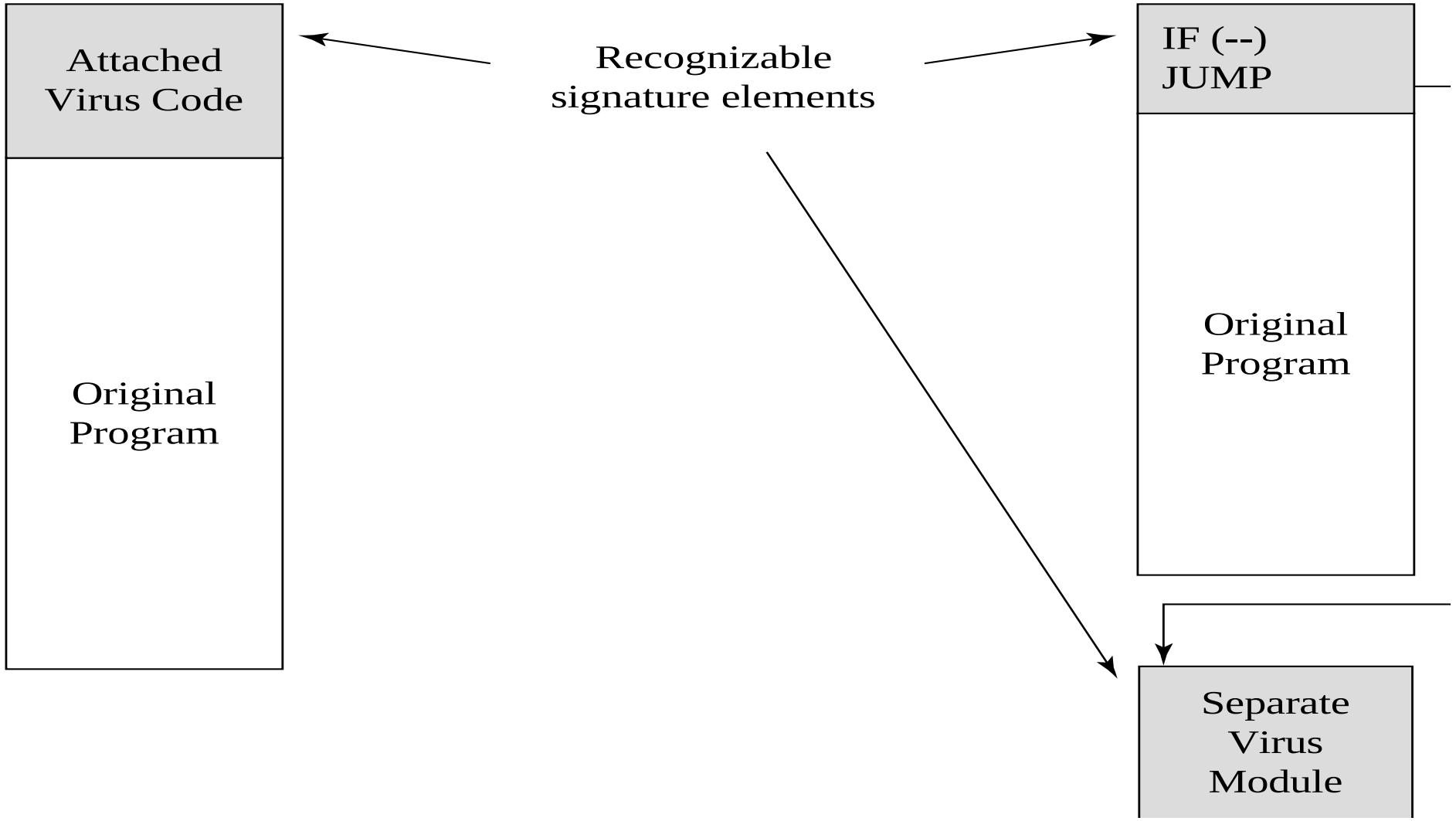
Virus Effects

Virus Effect	How It Is Caused
Attach to executable program	· Modify file directory · Write to executable program file
Attach to data or control file	· Modify directory · Rewrite data · Append to data · Append data to self
Remain in memory	· Intercept interrupt by modifying interrupt handler address table · Load self in non-transient memory area
Infect disks	· Intercept interrupt · Intercept operating system call (to format disk, for example) · Modify system file · Modify ordinary executable program
Conceal self	· Intercept system calls that would reveal self and falsify result · Classify self as “hidden” file
Spread infection	· Infect boot sector · Infect systems program · Infect ordinary program · Infect data ordinary program reads to control its execution
Prevent deactivation	· Activate before deactivating program and block deactivation · Store copy to reinfect after deactivation

Virus Detection

- Virus scanners look for signs of malicious code infection using signatures in program files and memory
- Traditional virus scanners have trouble keeping up with new malware—detect about 45% of infections
- Detection mechanisms:
  - Known string patterns in files or memory
  - Execution patterns
  - Storage patterns

Virus Signatures



Malware Detection

- Three common methods
  - Signature detection
  - Change detection
  - Anomaly detection
- We'll briefly discuss each of these
  - And consider advantages and disadvantages of each

Signature Detection

- A **signature** is a string of bits found in software (or could be a hash value)
- Suppose that a virus has signature 0x23956a58bd910345
- We can search for this signature in all files
- If we find the signature are we sure we've found the virus?
  - No, same signature could appear in other files
  - But at random, chance is very small: $1/2^{64}$
  - Software is not random, so probability is higher

Signature Detection

- Advantages
  - Effective on “traditional” malware
  - Minimal burden for users/administrators
- Disadvantages
  - Signature file can be large (10,000’s)…
  - …making scanning slow
  - Signature files must be kept up to date
  - Cannot detect unknown viruses
  - Cannot detect some new types of malware
- By far the most popular detection method

Change Detection

- Viruses must live somewhere on system
- If we detect that a file has changed, it may be infected
- How to detect changes?
  - Hash files and (securely) store hash values
  - Recompute hashes and compare
  - If hash value changes, file **might** be infected
  - Check for oligomorphism and polymorphism

Change Detection

- Advantages
  - Virtually no false negatives
  - Can even detect previously unknown malware
- Disadvantages
  - Many files change — and often
  - Many false alarms (false positives)
  - Heavy burden on users/administrators
  - If suspicious change detected, then what?
  - Might still need signature-based system

Anomaly Detection

- Monitor system for anything “unusual” or “virus-like” or potentially malicious
- What is unusual?
  - Files change in some unusual way
  - System misbehaves in some way
  - Unusual network activity
  - Unusual file access, etc., etc., etc.
- But must first define “normal”
  - And normal can change!

Anomaly Detection

- Advantages
  - Chance of detecting unknown malware
- Disadvantages
  - Unproven in practice
  - Trudy can make abnormal look normal (go slow)
  - Must be combined with another method (such as signature detection)
- Also popular in intrusion detection (IDS)
- A difficult unsolved (unsolvable?) problem
  - As difficult as AI?

Countermeasures for Users

- Use software acquired from reliable sources
- Test software in an isolated environment
- Only open attachments when you know them to be safe
- Treat every website as potentially harmful
- Create and maintain backups

Countermeasures for Developers

- Software Engineering Techniques
- Information hiding - Information hiding: describing what a module does, not how
- Modularity – Coupling and cohesion
- Mutual Suspicion
- Confinement - possible damage does not spread to other parts of a system
- Simplicity - Complexity is often the enemy of security
- Genetic Diversity - Diversity reduces the number of targets susceptible to one attack type
- Testing

Code Testing

- Unit testing
- Integration testing
- Function testing
- Performance testing
- Acceptance testing
- Installation testing
- Regression testing
- Penetration testing

Countermeasure Specifically for Security – Design Principles for Security

- Least privilege
- Economy of mechanism
- Open design
- Complete mediation
- Permission based
- Separation of privilege
- Least common mechanism
- Ease of use

Countermeasure Specifically for Security – Design Principles for Security

- Penetration Testing for Security
- Proofs of Program Correctness
- Validation
- Defensive Programming - Program designers and implementers need not only write correct code but must also anticipate what could go wrong.
- Trustworthy Computing Initiative - all developers undergo security training, and secure software development practices were instituted throughout the company

Countermeasures that Don’t Work

- Penetrate-and-Patch - Penetrate-and-patch fails because it is hurried, misses the context of the fault, and focuses on one failure, not the complete system.

- Security by Obscurity - Security by obscurity is the belief that a system can be secure as long as nobody outside its implementation group is told anything about its internal mechanisms.

Summary

- Buffer overflow attacks can take advantage of the fact that code and data are stored in the same memory in order to maliciously modify executing programs
- Programs can have a number of other types of vulnerabilities, including off-by-one errors, incomplete mediation, and race conditions
- Malware can have a variety of harmful effects depending on its characteristics, including resource usage, infection vector, and payload
- Developers can use a variety of techniques for writing and testing code for security