When is a program secure?

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

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

What is a software vulnerability?

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

- There are a lot of types of vulnerabilities, but among the most classic and important are vulnerabilities that violate “control flow integrity”
  - Translation: lets attacker run code of their choosing on your computer

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

- Today we begin our dive into low level details of how exploits work
  - How can a remote attacker get your machine to execute their code?

- Our threat model
  - Victim code is handling input that comes from across a security boundary
    - Examples:
      - Image viewer, word processor, web browser
      - Other examples?
    - We want to protect integrity of execution and confidentiality of internal data from being compromised by malicious and highly skilled users of our system.

- Simplest example: buffer overflow
  - Provide input that “overflows” the memory the program has allocated for it
Lecture Objectives

▪ Understand how buffer overflow vulnerabilities can be exploited
▪ Identify buffer overflow vulnerabilities in code and assess their impact
▪ Avoid introducing buffer overflow vulnerabilities during implementation
Buffer Overflow

- **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

```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);
}
```
Old school: The Trouble With **strc***()*

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

- What’s the problem with libc string functions?
  - Neither `strcpy()` nor `strcat()` validate that the destination string has enough space to fit the source string.
  - They also provide no mechanism to signal an error.

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

- These functions are considered unsafe across the industry.
Old school: Replacing `strc*()`

```c
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.
Old school: The Trouble With strncpy*()

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

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

- Aside: if you **must** manipulate strings in C, then strl*() functions are much safer
  - Guarantees NULL termination and doesn’t require complex address arithmetic
But its not just a “C strings” problem

- C string functions are particularly egregious, but there are lots of other ways a local buffer can be overflowed
  - Memcpy/bcopy, arrays, pointer arithmetic, bad casts, etc...
Ok, sure 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
First, some context

- How memory is laid out in a process
- How C arrays work
- How C function calls work
How process memory is laid out (Linux 32bit traditional, simplified)

- Stack
  - Locals, call stack

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

- Data segment (globals, statics)
  - .data
  - .bss

- Text segment
  - Executable code
How do C arrays work?

▪ What’s the abstraction?

▪ What’s the reality?
  – What happens if you try to write past the end of an array in C/C++
  – What does the spec say?
  – What happens in most implementations?
Understanding Function Calls

- How does a function call work?
  - What’s the abstraction?
    ```
    bar() {
      foo();
    }
    ```
  - What’s the reality?
    - Where does the memory for i from `foo()` come from?
    - How does the called function know where to return to?
    - Where is the return address stored?

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

- Stack divided into **frames**
  - Each frame stores locals and args to called functions

- **Stack pointer** points to the top of the stack
  - x86: stack grows down (from high to low addresses)
  - x86: stored in %esp register

- **Frame pointer** points to caller’s frame on the stack
  - Also called (by Intel) the base pointer
  - x86: Stored in %ebp register
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/](https://godbolt.org/)

Note: in x86, the `leave` instruction =

```
  mov esp, ebp
  pop ebp
```

```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;
}
```
Quick aside: Intel vs AT&T syntax

- I’ve been using Intel asm syntax
  - instruction dst src
  - mov esp, ebp

- gdb uses AT&T asm syntax (also the Aleph One article)
  - Instruction src dst
  - movl %ebp, %esp

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

- If you see a register prefixed with “%” you’re dealing with AT&T syntax
Back to buffer overflows...

- So... consider this program ->
- It takes input from the console and then prints it followed by “is nice\n”
- How long can your input be?
- What happens if its longer?

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

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

- Mixing control and user data creates an opportunity for attackers
- 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
    - Overflow can happen outside of current function’s frame
  - Exception control data
Smashing The Stack: getting lucky

- Overwriting **local variables** or **function arguments**
  - Effect depends on variable semantics and usage
  - Generally anything that influences future execution path is a promising target
  - Typical problem cases:
    - Variables that store result of a security check
      - Eg. isAuthenticated, isValid, isAdmin, etc.
    - Variables used in security checks
      - Eg. buffer_size, etc.
    - Data pointers
      - Potential for further memory corruption
    - Function pointers
      - Direct transfer of control when function is called through overwritten pointer
Smashing The Stack: control data

- 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
      - Shellcode! (coming up)
    - Reminder: there’s nothing that distinguished data from code
      - All data (including input) will be interpreted as code if the processor tries to transfer control there
  - **Game over**
Smashing The Stack: off by one errors

- But what if you can only overwrite **one word** or **one byte**?
  - Seems hard to exploit no?

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

- What to do after we figure out how to seize control of the instruction pointer?
- Ideally, redirect to our own code!
- But what should that code be?

- Spawning a shell would provide us with full privileges of the victim process
  - Hence, “shellcode”
Shellcode

- How to spawn a shell?

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

- Just need to call `execve` with the right arguments
  - `execve("/bin/sh", argv, NULL)`
Shellcode

- Note the tricks Aleph One uses:
  - Writing shellcode in C
    - Compile and run in debugger to review object code
    - Adjust references to strings, etc.

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

- Note the tricks Aleph One uses:
  - Inline assembly to use gcc to translate from assembly to object code
    - Compile and run in debugger to review object code
  - Using a call instruction to infer the address of payload on the stack
    - call will push the address of the next word onto the stack as a return address

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

http://phrack.org/issues/49/14.html
Shellcode

- Note the tricks Aleph One uses:
  - Testing shellcode standalone
    - Encode shellcode into a data buffer
    - Set the return address on the stack to point to your shellcode
  - Eliminating 0x00 from the shellcode
    - Find alternate instruction representations
  - Using a NOP sled
    - Relaxes constraints on guessing the exact location of the shellcode to put into the overwritten return address
    - Jump to somewhere in NOP sled and slide down to shellcode

```c
void main() {
    int *ret;
    ret = (int *)&ret + 2;
    (*ret) = (int)shellcode;
}
```

```c
char shellcode[] = "xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
    "\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
    "\xb8\x01\x00\x00\xb0\x00\xa4\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
    "\xb0\x0b"
    "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x8d\x6f\x6e\x2f\x73\x68\x00\x89\xec\x5d\x43";
```

```
char shellcode[] = "xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
    "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\x40\xcd"
    "\x80\xe0\x6c\xff\xff\xff/b1n/sh/"
```

http://phrack.org/issues/49/14.html
Shellcode

- That works well for local attacks
  - When the victim is another process on the same machine

- What about remote attacks?

- Similar concept, just a few more system calls in the shellcode
  - Reverse
    - Connect back to your malicious server and present a remote shell
  - Bind
    - Open a port and wait for connections, present shell
  - Reuse
    - Re-use existing connection
Common Buffer Overflow Patterns

- Spotting buffer overflow bugs in code
  - Missing Check
  - Avoidable Check
  - Wrong Check
Buffer Overflow Code Patterns

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

- Example
  - fingerd

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>

int main(int argc, char *argv[]) {
    char *argv[] = {argv[0], NULL};
    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

```c
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;
```
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
Common Buffer Overflow Patterns

▪ Thinking like an attacker:
  – Missing Check
    ▪ Does the code perform bounds checking on memory access?
  – Avoidable Check
    ▪ Is the test invoked along every path leading up to actual access?
  – Wrong Check
    ▪ Is the test correct? Can the test itself be attacked?

▪ Generic input validation patterns
  – Applicable beyond just buffer overflows
Addressing Buffer Overflows

- The best way to deal with any bug is not to have it in the first place.
  - Use memory-safe languages.
  - Train the developers to write secure code and provide them with tools that make it easier to do so.

- Language choice might not be an option (it frequently isn’t) and people still make mistakes. So, we must also be able to find these bugs and fix them.
  - Manual code reviews, static analysis, adversarial testing, etc.
  - More on this later in the course...

- Failing all of the above, make remaining bugs harder to exploit.
  - Introduce countermeasures that make reliable exploitation harder or mitigate the impact
  - Lecture after next
Review

- An attacker can direct the execution of your program by manipulating input data it acts on.
- Assume input can be malicious. **Always** validate lengths and bounds before accessing arrays.
- Separate control data from user data where possible
- Default ways of doing something are often insecure. Investigate security aspects of tools, frameworks, libraries, APIs, that you are using and understand how to use them safely.
Review

- Writing past the bounds of a buffer can have severe consequences.
- Overwriting the return address
  - Upon function return, control is transferred to an attacker-chosen address
  - Arbitrary code execution
    - Attacker can re-direct to their own code
Additional Resources

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

- More in future lectures...
Additional Resources

- John Regehr’s blog on undefined behavior
  - https://blog.regehr.org/page/2?sr=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
For next time

- Beyond the basic buffer overflow: integers, heap, format strings (interpreters)
- Read *Memory Errors: The Past, the Present, and the Future* by Victor van der Veen, Nitish dutt-Sharma, Lorenzo Cavallaro, and Herbert Bos
Next Lecture...

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