Logistics

• Things Due:
  – Homework #1: Due Wednesday

• Things Releasing:
  – Lab 1 TODAY
Last time...

- Threat models
  - Assets
  - Adversaries
  - Vulnerabilities
  - Threats
  - Risks
SOFTWARE SECURITY
Bugs, Vulnerabilities, and Exploits

• Bug
  – Not working quite right

• Vulnerability
  – A malfunction that can be used for an adversary’s goals

• Exploit
  – The mechanical set of operations to make use of a vulnerability
Aside: The Weird Machine

• An exploit can also be considered a *program* for a *weird machine*

• If you are more formally-inclined, check out:
  – [https://www.cs.dartmouth.edu/~sergey/wm/](https://www.cs.dartmouth.edu/~sergey/wm/)

• We’ll come back to this later in the course
Adversarial Failures

• Software bugs are bad
  – Consequences can be serious

• Even worse when an intelligent adversary wishes to exploit them!
  – Intelligent adversaries: Force bugs into “worst possible” conditions/states
  – Intelligent adversaries: Pick their targets
Many types of vulnerability
Memory Corruption Bugs

• **Buffer overflows bugs**: Big class of bugs
  – Normal conditions: Can sometimes cause systems to fail
  – Adversarial conditions: Attacker able to violate security of your system (control, obtain private information, ...)

• Stack, Heap both possibilities
BUFFER OVERFLOWS
A Bit of History: Morris Worm

• Worm was released in 1988 by Robert Morris
  – Graduate student at Cornell, son of NSA chief scientist
  – Convicted under Computer Fraud and Abuse Act,
    • 3 years probation and 400 hours of community service

• Worm was intended to propagate slowly and harmlessly measure the size of the Internet

• Due to a coding error, it created new copies as fast as it could and overloaded infected machines

• $10-100M worth of damage
Morris Worm and Buffer Overflow

• One of the worm’s propagation techniques was a buffer overflow attack against a vulnerable version of fingerd on VAX systems
  – By sending special string to finger daemon, worm caused it to execute code creating a new worm copy

Buffer overflows remain a common source of vulnerabilities and exploits today!
  (Especially in embedded systems.)
Aside: Famous Internet Worms

- **Morris worm (1988):** overflow in `fingerd`
  - 6,000 machines infected
- **CodeRed (2001):** overflow in MS-IIS server
  - 300,000 machines infected in 14 hours
- **SQL Slammer (2003):** overflow in MS-SQL server
  - 75,000 machines infected in **10 minutes** (!!)
- **Sasser (2005):** overflow in Windows LSASS
  - Around 500,000 machines infected
... And More

- **Conficker (2008-09):** overflow in Windows RPC
  - Around 10 million machines infected (estimates vary)
- **Stuxnet (2009-10):** several zero-day overflows + same Windows RPC overflow as Conficker
  - Windows print spooler service
  - Windows LNK shortcut display
  - Windows task scheduler
- **Flame (2010-12):** same print spooler and LNK overflows as Stuxnet
  - Targeted cyberspoionage virus
- **These days, worms are uncommon**
... And More

- **Embedded systems**
  - E.g., our automotive work
- **Formative and foundational for software security**
Attacks on Memory Buffers

- **Buffer** is a pre-defined data storage area inside computer memory (stack or heap)

- **Typical situation:**
  - A function takes some input that it writes into a **pre-allocated buffer**.
  - The developer **forgets to check** that the size of the input isn’t larger than the size of the buffer.
  - *Uh oh.*
    - “Normal” bad input: crash
    - “Adversarial” bad input: take control of execution
Suppose Web server contains this function

```c
void func(char *str) {
    char buf[126];
    ...
    strcpy(buf, str);
    ...
}
```

- No bounds checking on `strcpy()`
- If `str` is longer than 126 bytes
  - Program may crash
  - Attacker may change program behavior
Example: Changing Flags

• Suppose Web server contains this function

```c
void func(char *str) {
    byte auth = 0;
    char buf[126];
    ...
    strcpy(buf, str);
    ...
}
```

• Authenticated variable non-zero when user has extra privileges

• Morris worm also overflowed a buffer to overwrite an authenticated flag in fingerd
Memory Layout

- **Text region**: Executable code of the program
- **Heap**: Dynamically allocated data
- **Stack**: Local variables, function return addresses; grows and shrinks as functions are called and return
Stack Buffers

• Suppose Web server contains this function:

```c
void func(char *str) {
    char buf[126];
    strcpy(buf, str);
}
```

• When this function is invoked, a new frame (activation record) is pushed onto the stack.

Allocate local buffer (126 bytes reserved on stack)
Copy argument into local buffer

Local variables

Execute code at this address after func() finishes
What if Buffer is Overstuffed?

• Memory pointed to by str is copied onto stack...

```c
void func(char *str) {
    char buf[126];
    strcpy(buf,str);
}
```

• If a string longer than 126 bytes is copied into buffer, it will overwrite adjacent stack locations.

strncpy does NOT check whether the string at *str contains fewer than 126 characters

This will be interpreted as return address!
Executing Attack Code

• Suppose buffer contains attacker-created string
  – For example, str points to a string received from the network as the URL

• When function exits, code in the buffer will be executed, giving attacker a shell ("shellcode")
  – Root shell if the victim program is setuid root
Buffer Overflows Can Be Tricky...

• Overflow portion of the buffer must contain **correct address of attack code** in the RET position
  – The value in the RET position must point to the beginning of attack assembly code in the buffer
    • Otherwise application will (probably) crash with segfault
  – **Attacker must correctly guess in which stack position his/her buffer will be when the function is called**
Problem: No Bounds Checking

• `strcpy` does **not** check input size
  – `strcpy(buf, str)` simply copies memory contents into `buf` starting from `*str` until “\0” is encountered, ignoring the size of area allocated to `buf`

• Many C library functions are unsafe
  – `strcpy(char *dest, const char *src)`
  – `strcat(char *dest, const char *src)`
  – `gets(char *s)`
  – `scanf(const char *format, ...)`
  – `printf(const char *format, ...)"
Does Bounds Checking Help?

• `strncpy(char *dest, const char *src, size_t n)`
  – If `strncpy` is used instead of `strcpy`, no more than `n` characters will be copied from `*src` to `*dest`
    • Programmer has to supply the right value of `n`

• Potential overflow in `htpasswd.c` (Apache 1.3):

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

• Published fix:

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

Copies username ("user") into buffer ("record"), then appends ":" and hashed password ("cpw"
Breakout Activity

Gradescope!
Misuse of strncpy in htpasswd “Fix”

• Published “fix” for Apache htpasswd overflow:

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

**Diagram:**

- MAX_STRING_LEN bytes allocated for record buffer
- Put up to MAX_STRING_LEN-1 characters into buffer
- Put "":
- Again put up to MAX_STRING_LEN-1 characters into buffer
What About This?

- Home-brewed range-checking string copy

```c
void mycopy(char *input) {
  char buffer[512]; int i;
  for (i=0; i<=512; i++)
    buffer[i] = input[i];
}

void main(int argc, char *argv[]) {
  if (argc==2)
    mycopy(argv[1]);
}
```
Breakout Activity

Gradescope again!
Off-By-One Overflow

• Home-brewed range-checking string copy

```c
void mycopy(char *input) {
    char buffer[512]; int i;

    for (i=0; i<=512; i++)
        buffer[i] = input[i];
}
```

```c
void main(int argc, char *argv[]) {
    if (argc==2)
        mycopy(argv[1]);
}
```

• 1-byte overflow: can’t change RET, but can change pointer to previous stack frame...

This will copy 513 characters into buffer. Oops!
Frame Pointer Overflow

Fake FP | Fake RET
-------|-------
buf | Saved FP | ret/IP | str
Local variables | Args | Caller’s frame
Addr 0xFF...F

ATTACK CODE
Another Variant: Function Pointer Overflow

- C uses **function pointers** for callbacks: if pointer to F is stored in memory location P, then one can call F as (*P)(...)

![Diagram of Function Pointer Overflow]

- Buffer with attacker-supplied input string
- Callback pointer
- attack code
- overflow
- Legitimate function F (elsewhere in memory)
Other Overflow Targets

• Format strings in C
  – We’ll walk through this one next time

• Heap management structures used by malloc()
  – More details in section
  – Techniques have changed wildly over time

• These are all attacks you can look forward to in Lab #1 😊