CSE 484 / CSE M 584: Defenses, Software Security, Buffer Overflows

Fall 2022

Franziska (Franzi) Roesner
franzi@cs

UW Instruction Team: David Kohlbrenner, Yoshi Kohno, Franziska Roesner. Thanks to Dan Boneh, Dieter Gollmann, Dan Halperin, John Manferdelli, John Mitchell, Vitaly Shmatikov, Bennet Yee, and many others for sample slides and materials ...
Announcements

• Things Due:
  – Ethics Form: Due today!
  – Homework #1: Due Friday
  – Research Readings (CSE M 584): Due Thursday (and every Thursday thereafter)
TOWARDS DEFENSES
Approaches to Security

• Prevention
  – Stop an attack
• Detection
  – Detect an ongoing or past attack
• Response and Resilience
  – Respond to / recover from attacks

• The threat of a response may be enough to deter some attackers
Whole System is Critical

• Securing a system involves a whole-system view
  – Cryptography
  – Implementation
  – People
  – Physical security
  – Everything in between

• This is because “security is only as strong as the weakest link,” and security can fail in many places
  – No reason to attack the strongest part of a system if you can walk right around it.
Whole System is Critical

• Securing a system involves a whole-system view
  – Cryptography
  – Implementation
  – People
  – Physical security
  – Everything in between

• This is because “security is only as strong as the weakest link,” and security can fail in many places
  – No reason to attack the strongest part of a system if you can walk right around it.
Whole System is Critical

Securing a system involves a whole-system view—Cryptography—Implementation—People—Physical security—Everything in between.

This is because "security is only as strong as the weakest link," and security can fail in many places—No reason to attack the strongest part of a system if you can walk right around it.
Whole System is Critical

Securing a system involves a whole-system view—Cryptography—Implementation—People—Physical security—Everything in between. This is because “security is only as strong as the weakest link,” and security can fail in many places—no reason to attack the strongest part of a system if you can walk right around it.
Attacker’s Asymmetric Advantage
Attacker’s Asymmetric Advantage

- Attacker only needs to win in one place
- Defender’s response: Defense in depth
From Policy to Implementation

• After you’ve figured out what security means to your application, there are still challenges:
  – Requirements bugs and oversights
    • Incorrect or problematic goals
  – Design bugs and oversights
    • Poor use of cryptography
    • Poor sources of randomness
    • ...
  – Implementation bugs and oversights
    • Buffer overflow attacks
    • ...
  – Is the system usable?
Many Participants

• Many parties involved
  – System developers
  – Companies deploying the system
  – The end users
  – The adversaries (possibly one of the above)

• Different parties have different goals
  – System developers and companies may wish to optimize cost
  – End users may desire security, privacy, and usability
  – Different users/stakeholders may have different needs
  – The relationship between these goals is quite complex (e.g., will customers choose features or security?) (e.g., are there “non-obvious” stakeholders?)
Better News

• There are a lot of defense mechanisms
  – We’ll study some, but by no means all, in this course

• It’s important to understand their limitations
  – “If you think cryptography will solve your problem, then you don’t understand cryptography... and you don’t understand your problem” -- Bruce Schneier
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
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
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 (in 1988)
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.)
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
Stack Buffers

• 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

<table>
<thead>
<tr>
<th>Text region</th>
<th>Heap</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr 0x00...0</td>
<td></td>
<td>Addr 0xFF...F</td>
</tr>
</tbody>
</table>

![Memory Layout Diagram]
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.

[Diagram showing stack layout with labels for buf, Saved FP, ret/IP, str, Local variables, Args, and Caller’s frame.]

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.

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

This will be interpreted as return address!

Adj 0xFF...F
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)
  – For **strncpy** (unlike **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, " :" )
  strcat (record, cpw, MAX_STRING_LEN-1 );
  ```

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

Canvas -> Quizzes -> Oct 3

(This is the first one that will be graded.
Reminder that you have 5 “freebies” for the quarter.)
Misuse of strncpy in htpasswd “Fix”

- Published “fix” for Apache htpasswd overflow:

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

MAX_STRING_LEN bytes allocated for record buffer

- Contents of *user
- Contents of *cpw

- 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]);
}
```
In-Class Activity

Canvas -> Quizzes -> Oct 5
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];
}

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  ATTACK CODE

buf  Saved FP  ret/IP  str  Caller’s frame

Local variables  Args  Addr 0xFF...F
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:
- 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 later
• 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 😊