Software Security: Buffer Overflow Attacks

Autumn 2020

Franziska (Franzi) Roesner
franzi@cs.washington.edu

Thanks to Dan Boneh, Dieter Gollmann, Dan Halperin, Yoshi Kohno, Ada Lerner, John Manferdelli, John Mitchell, Vitaly Shmatikov, Bennet Yee, and many others for sample slides and materials...
Announcements

• Things Due:
  – Ethics form: Due Wednesday
  – Homework #1: Due Friday

• Office Hours:
  – Now scheduled, see course website
  – Via Zoom – find links on Canvas
  – Mine are right after class today (and all Mondays)

• Lab 1 coming up!
  – We will be sending out a sign-up form today
  – Section this week will be very important for lab 1

• Zoom Breakouts
  – You can join self-selected “Zoom Breakout” groups in Canvas, I will start using them Wednesday – keep scrolling in Canvas until that group set loads
TOWARDS DEFENSES
Approaches to Security

• Prevention
  – Stop an attack

• Detection
  – Detect an ongoing or past attack

• Response
  –Respond to 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:
  - 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
    • Incorrect or problematic goals
  – Design bugs
    • Poor use of cryptography
    • Poor sources of randomness
    • ...
  – Implementation bugs
    • 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
  – But the relationship between these goals is quite complex
    (will customers choose features or security?)
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
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

• 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, ...)

10/5/20
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
  – Now an EECS professor at MIT

• 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

• $10\text{–}100\text{M worth of damage}$
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.)
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) {
    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
```

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.

```c
strcpy 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

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

• Many C library functions are unsafe
  – \texttt{strcpy(char *dest, const char *src)}
  – \texttt{strcat(char *dest, const char *src)}
  – \texttt{gets(char *s)}
  – \texttt{scanf(const char *format, \ldots)}
  – \texttt{printf(const char *format, \ldots)}
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):

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

- Published fix:

```
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

Canvas -> Quizzes -> Oct 5

(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, "":"\nstrncat(record, cpw, MAX_STRING_LEN - 1);
```

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