Software Security: Buffer Overflow Attacks

Spring 2020

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

• Where to find what
  – **Course website:**
    • Schedule (with slide decks, assignment links)
    • Assignments (all homeworks, labs, research readings, final project)
  – **Canvas:**
    • Assignment / Lab / Readings / Project turn-in
    • In-class Activities (“Quizzes”)
    • Zoom links and cloud recordings
  – **Ed:**
    • Discussion board (questions, discussion)

• **Things Due:**
  – **Ethics form:** Due next Wednesday (4/8)
  – **Homework #1:** Due next Friday (4/10)
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
SOFTWARE SECURITY
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, ...)
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

• $10-100M 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.)
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.
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);
  }
  ```

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

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

  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);
  ```
  Copies username (“user”) into buffer (“record”), then appends “:” and hashed password (“cpw”)
- Published fix:
  ```c
  strncpy(record,user,MAX_STRING_LEN-1);
  strcat(record,":" )
  strcat(record,cpw,MAX_STRING_LEN-1);
  ```
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

Canvas -> Quizzes -> April 3

(This one still won’t be graded. We will send out an update via email about how we will be counting “in-class” activities before Monday.)

https://canvas.uw.edu/courses/1371936/quizzes/1234443
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);
```

MAX_STRING_LEN bytes allocated for record buffer

- Put up to MAX_STRING_LEN-1 characters into buffer
- Put ":" character
- 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]);
}
```
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]);
}
```

This will copy 513 characters into buffer. Oops!

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