CSE 484 / CSE M 584: Computer Security and Privacy

Autumn 2019

Tadayoshi (Yoshi) Kohno
yoshi@cs.Washington.edu

Thanks to Dan Boneh, Dieter Gollmann, Dan Halperin, John Manferdelli, John Mitchell, Franzi Roesner, Vitaly Shmatikov, Bennet Yee, and many others for sample slides and materials ...

Announcements

- Day Before Thanksgiving: Alternate Video Lesson (e.g., use to support your final project)
- Final Project: Please see information online
- My “Office Hours”:
  - This Wednesday, 11:30am, in CSE1 403, for group discussion, then moves to CSE2 307
  - Next Wednesday, 12:30pm, in CSE1 403, for group discussion, then moves to CSE2 307
- Quiz Section This Week: Workshop / Extended Office Hours
- Quiz Section Next Week: Try Target 5 in Advance

Announcements

- Format String Vulnerabilities, Other Exploits, and Course Structure: Don't worry if lectures alone leave open questions
- Recall themes / structure of course
  - Lectures: Big picture, key concepts, provide foundations, enable + provide tools for deeper learning through labs
  - Labs: Investigative opportunities for deeper technical explorations; *lots of learning for this course happens while puzzling through assignments*

FTC on LifeLock (Oct 8, 2019 News)

FTC Sends Checks Totaling More Than $31 Million to LifeLock Customers

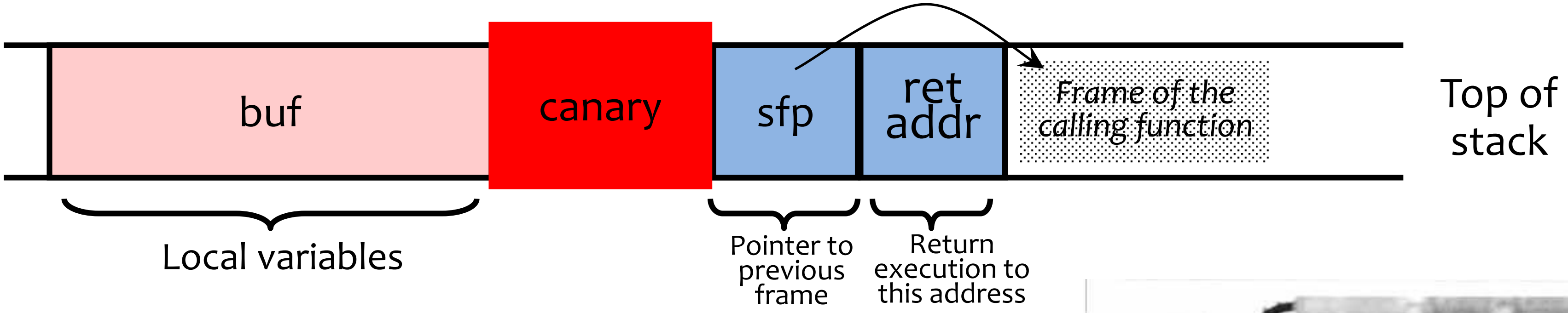
The refunds stem from a 2015 settlement LifeLock reached with the Commission, which alleged that from 2012 to 2014 **LifeLock violated an FTC order that required the company to secure consumers' personal information** and prohibited it from deceptive advertising. The FTC alleged, among other things, that **LifeLock failed to establish and maintain a comprehensive information security program to protect users' sensitive personal information**, falsely advertised that it protected consumers' sensitive data with the same high-level safeguards used by financial institutions, and falsely claimed it provided 24/7/365 alerts "as soon as" it received any indication a consumer's identity was being used.

Relates to class themes, including "what does security means", trust, levels of secruity

Back to Software Security

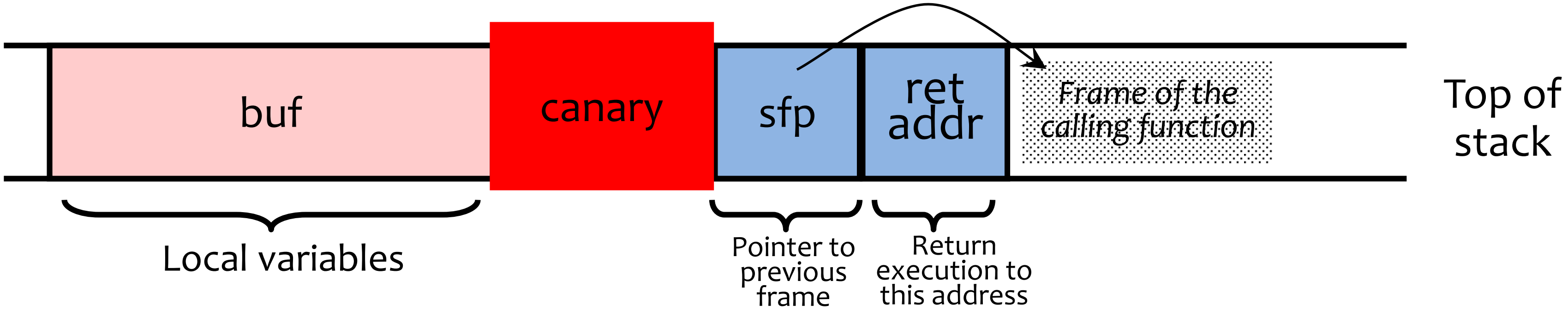
Run-Time Checking: StackGuard

- Embed "canaries" (stack cookies) in stack frames and verify their integrity prior to function return
  - Any overflow of local variables will damage the canary



Run-Time Checking: StackGuard

- Embed "canaries" (stack cookies) in stack frames and verify their integrity prior to function return
  - Any overflow of local variables will damage the canary



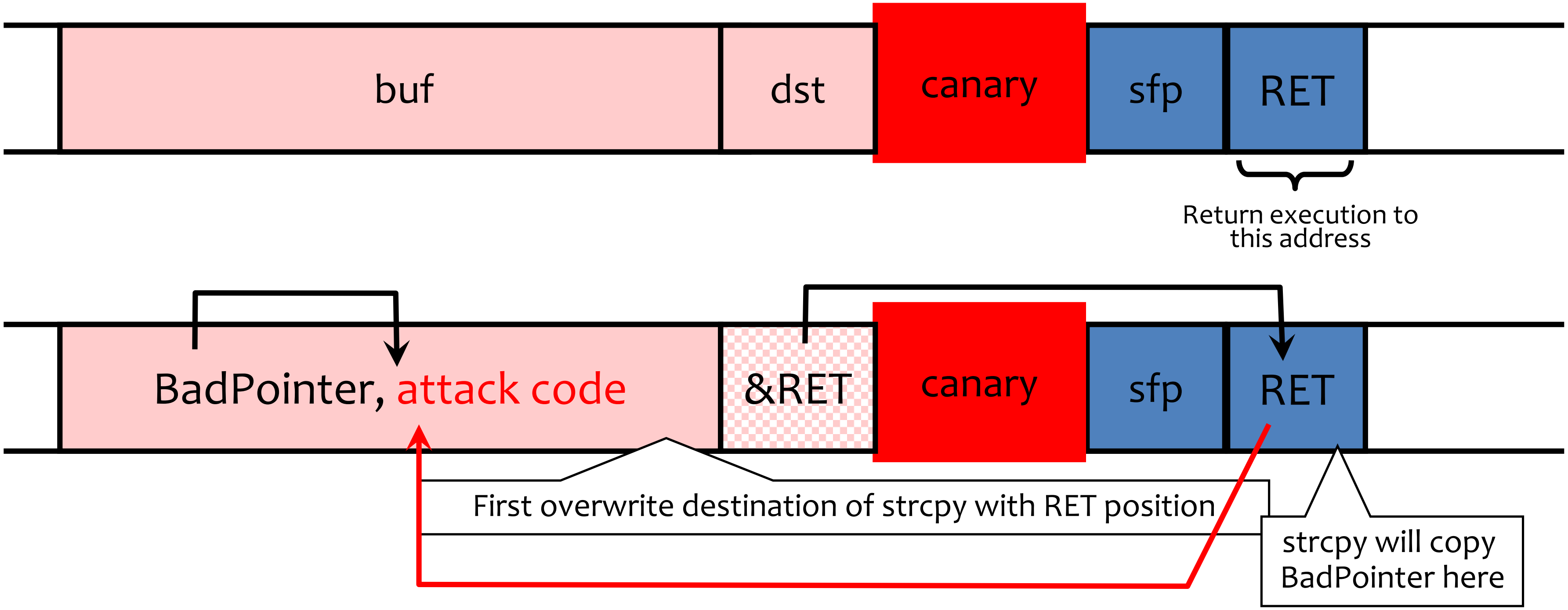
- Choose random canary string on program start
  - Attacker can't guess what the value of canary will be
- Terminator canary: "\0", newline, linefeed, EOF
  - String functions like strcpy won't copy beyond "\0"

StackGuard Implementation

- StackGuard requires code recompilation
- Checking canary integrity prior to every function return causes a performance penalty
  - For example, 8% for Apache Web server at one point in time
- StackGuard can be defeated
  - A single memory write where the attacker controls both the value and the destination is sufficient

Defeating StackGuard

- Suppose program contains strcpy(dst,buf) where attacker controls both dst and buf
  - Example: dst is a local pointer variable



First overwrite destination of strcpy with RET position

strcpy will copy BadPointer here

More on Defeating StackGuard

- Attacker sets buf to contain (first) a pointer to another region in buf with the attack code, and then (second) the attack code
- Attacker sets dst, to contain the address where RET is stored (recall the assumption that the attacker can also set dst)
- When the strcpy happens, memory beginning at the address of RET is overwritten with the contents of buf
  - This puts "BadPointer" in the location of RET
  - Recall that "BadPointer" is a value for the address at which the attack code starts (in buf)
- Can you think of other approaches?

ASLR: Address Space Randomization

- Randomly arrange address space of key data areas for a process
  - Base of executable region
  - Position of stack
  - Position of heap
  - Position of libraries
- Introduced by Linux PaX project in 2001
- Adopted by OpenBSD in 2003
- Adopted by Linux in 2005

ASLR: Address Space Randomization

- Deployment (examples)
  - Linux kernel since 2.6.12 (2005+)
  - Android 4.0+
  - iOS 4.3+ ; OS X 10.5+
  - Microsoft since Windows Vista (2007) (not by default)
- Attacker goal: Guess or figure out target address (or addresses)
- ASLR more effective on 64-bit architectures

ASLR Issues

- NOP slides and heap spraying to increase likelihood for custom code (e.g., on heap)
- Brute force attacks or memory disclosures to map out memory on the fly
  - Disclosing a single address can reveal the location of all code within a library, depending on the ASLR implementation

Other Possible Solutions

- Use safe programming languages, e.g., Java
  - What about legacy C code?
  - (Though Java doesn't magically fix all security issues ☺)
- Static analysis of source code to find overflows
- Dynamic testing: "fuzzing"
- Modern compiler options, e.g., incorporate stack canaries

Fuzz Testing

- Generate "random" inputs to program
  - Sometimes conforming to input structures (file formats, etc.)
- See if program crashes
  - If crashes, found a bug
  - Bug may be exploitable
- Surprisingly effective

- Now standard part of development lifecycle

Beyond Buffer Overflows…

Another Type of Vulnerability

- Consider this code:

```
int openfile(char *path) {
    struct stat s;
    if (stat(path, &s) < 0)
        return -1;
    if (!S_ISRREG(s.st_mode)) {
        error("only allowed to regular files!");
        return -1;
    }
    return open(path, O_RDONLY);
}
```

- **Goal:** Open only regular files (not symlink, etc)
- What can go wrong?

TOCTOU (Race Condition)

- TOCTOU == Time of Check to Time of Use:

```
int openfile(char *path) {
    struct stat s;
    if (stat(path, &s) < 0)
        return -1;
    if (!S_ISRREG(s.st_mode)) {
        error("only allowed to regular files!");
        return -1;
    }
    return open(path, O_RDONLY);
}
```

- **Goal:** Open only regular files (not symlink, etc)
- Attacker can change meaning of path between stat and open (and access files he or she shouldn’t)

This TOCTOU Example

- In call to open, pass O_NOFOLLOW to not follow symbolic links
- Call fstat on open file descriptor
- …
- Nice reference: https://developer.apple.com/library/archive/documentation/Security/Conceptual/SecureCodingGuide/Articles/RaceConditions.html

Another Type of Vulnerability

- Consider this code:

```
char buf[80];
void vulnerable() {
     int len = read_int_from_network();
     char *p = read_string_from_network();
     if (len > sizeof buf) {
          error("length too large, nice try!");
          return;
     }
     memcpy(buf, p, len);
}
```

```
void *memcpy(void *dst, const void * src, size_t n);
```

```
typedef unsigned int size_t;
```

Implicit Cast

- Consider this code:

```
char buf[80];
void vulnerable() {
     int len = read_int_from_network();
     char *p = read_string_from_network();
     if (len > sizeof buf) {
          error("length too large, nice try!");
          return;
     }
     memcpy(buf, p, len);
}
```

If len is negative, may copy huge amounts of input into buf.

```
void *memcpy(void *dst, const void * src, size_t n);
```

```
typedef unsigned int size_t;
```

Another Example

```
size_t len = read_int_from_network();
char *buf;
buf = malloc(len+5);
read(fd, buf, len);
```

(from [www-inst.eecs.berkeley.edu—implflaws.pdf]())

Integer Overflow

```
size_t len = read_int_from_network();
char *buf;
buf = malloc(len+5);
read(fd, buf, len);
```

- What if len is large (e.g., len = 0xFFFFFFFF)?
- Then len + 5 = 4 (on many platforms)
- Result: Allocate a 4-byte buffer, then read a lot of data into that buffer.

(from www-inst.eecs.berkeley.edu—implflaws.pdf)

Password Checker

- Functional requirements
  - PwdCheck(RealPwd, CandidatePwd) should:
    - Return TRUE if RealPwd matches CandidatePwd
    - Return FALSE otherwise
  - RealPwd and CandidatePwd are both 8 characters long
- Implementation (like TENEX system)

```
PwdCheck(RealPwd, CandidatePwd)  // both 8 chars
   for i = 1 to 8 do
      if (RealPwd[i] != CandidatePwd[i]) then
         return FALSE
   return TRUE
```

- Clearly meets functional description

Attacker Model

```
PwdCheck(RealPwd, CandidatePwd)  // both 8 chars
    for i = 1 to 8 do
        if (RealPwd[i] != CandidatePwd[i]) then
            return FALSE
    return TRUE
```

- Attacker can guess CandidatePwds through some standard interface
- Naive: Try all 256^8 = 18,446,744,073,709,551,616 possibilities
- Better: Time how long it takes to reject a CandidatePasswd. Then try all possibilities for first character, then second, then third, ...
  - Total tries: 256*8 = 2048

Timing Attacks

- Assume there are no "typical" bugs in the software
  - No buffer overflow bugs
  - No format string vulnerabilities
  - Good choice of randomness
  - Good design
- The software may still be vulnerable to timing attacks
  - Software exhibits input-dependent timings
- Complex and hard to fully protect against

Other Examples

- Plenty of other examples of timings attacks
  - AES cache misses
    - AES is the “Advanced Encryption Standard”
    - It is used in SSH, SSL, IPsec, PGP, …
  - RSA exponentiation time
    - RSA is a famous public-key encryption scheme
    - It’s also used in many cryptographic protocols and products
  - Recently: Spectre and Meltdown