CSE 484 / CSE M 584: Computer Security and Privacy

Software Security: Miscellaneous

Fall 2016

Adam (Ada) Lerner
lerner@cs.washington.edu

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

Security Mindset Anecdote

- Craigslist

Security Mindset Anecdote

Raye had recently evicted a tenant and cleaned out the rental.

The ad posted last weekend welcomed people to take for free anything they wanted from the home. It has since been pulled from the site, but not before the residence was stripped of light fixtures, the hot water heater and the kitchen sink.

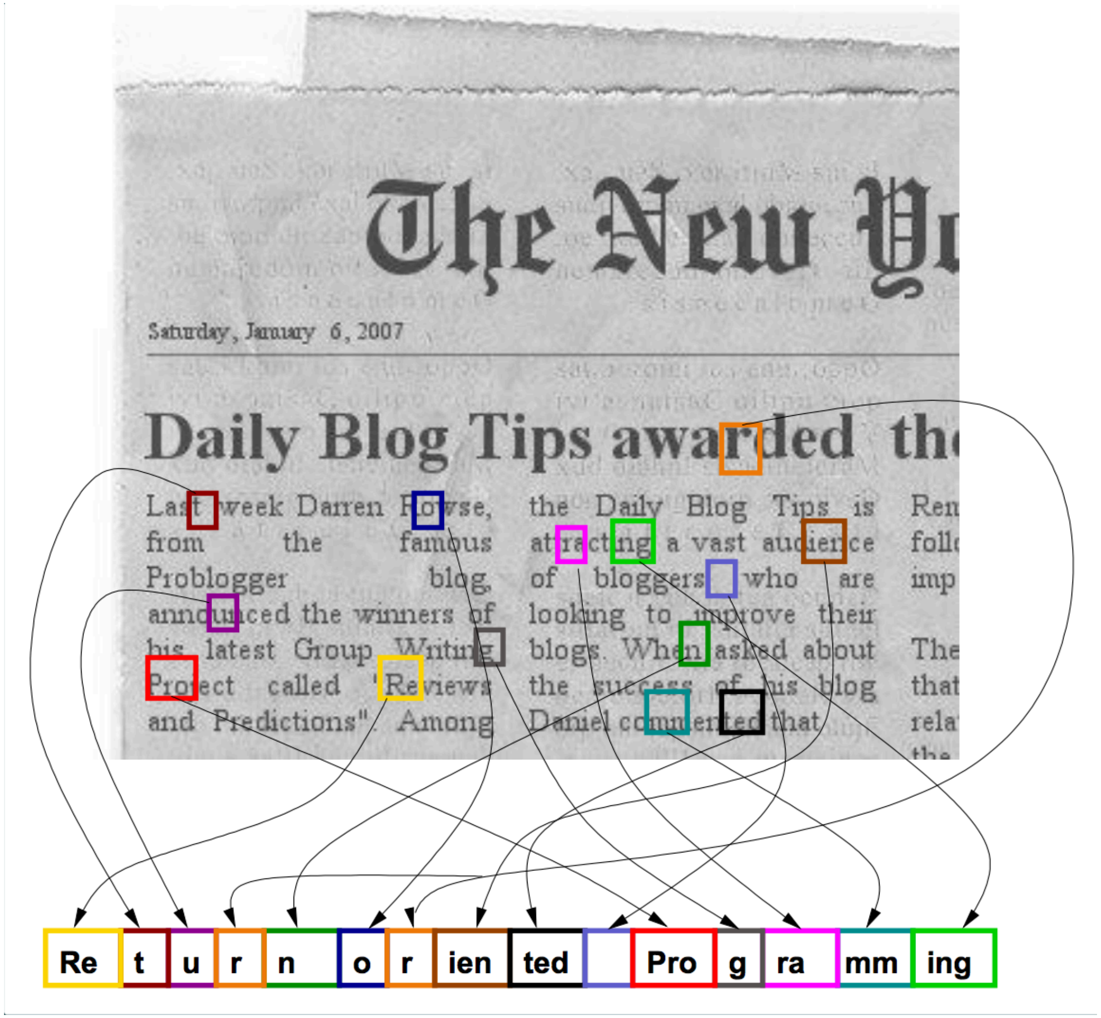
Neighbors said they saw strangers hauling items away, apparently looking for salvage material.

Even the front door and a vinyl window were pilfered, Raye said.

Looking Forward

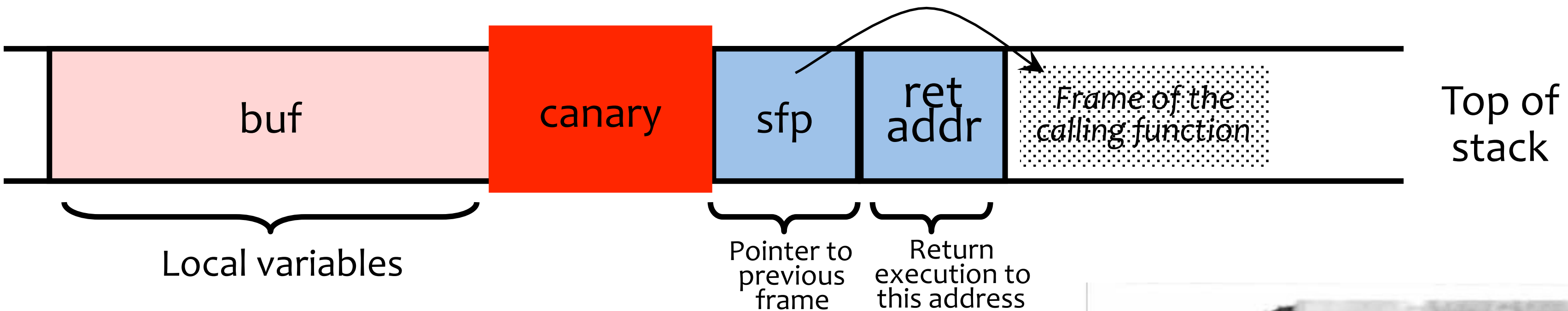
- Lab 1 checkpoint is due today
- Sploits 4-7 are due October 31 – **get working** – a crypto homework will be emerging before they're due!

Return-Oriented Programming



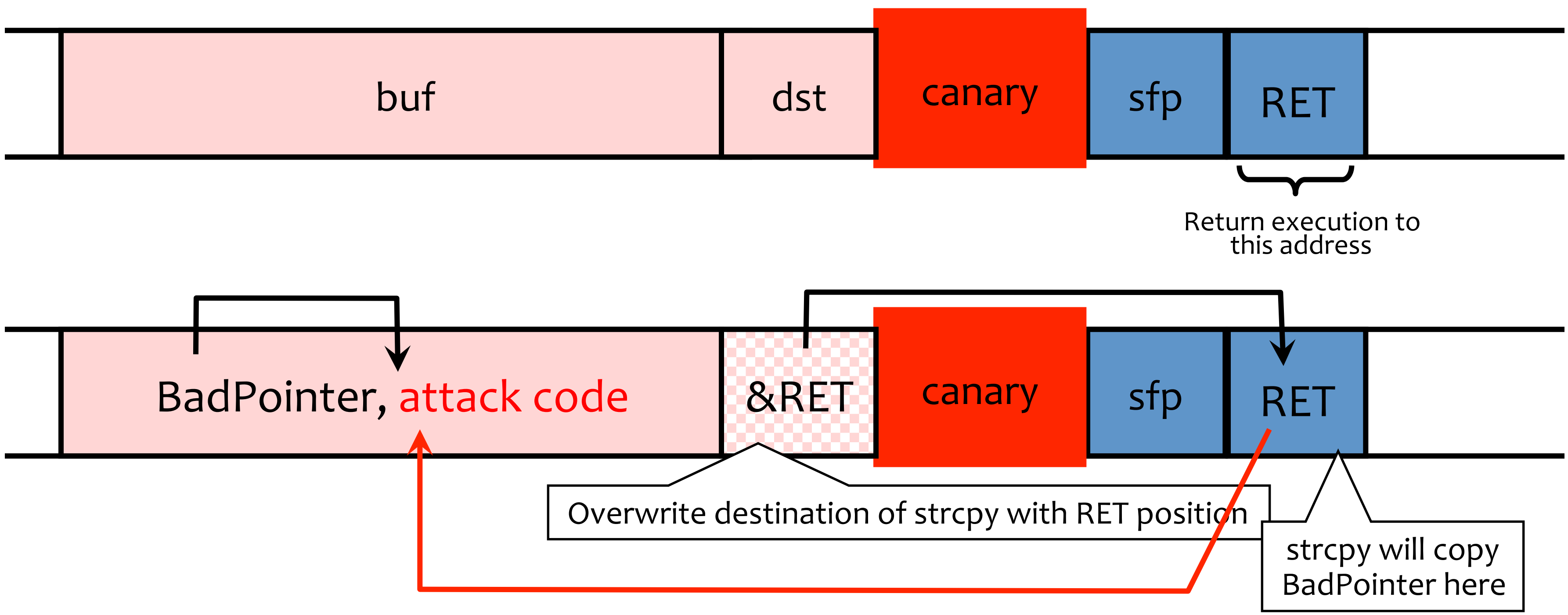
Run-Time Checking: StackGuard

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



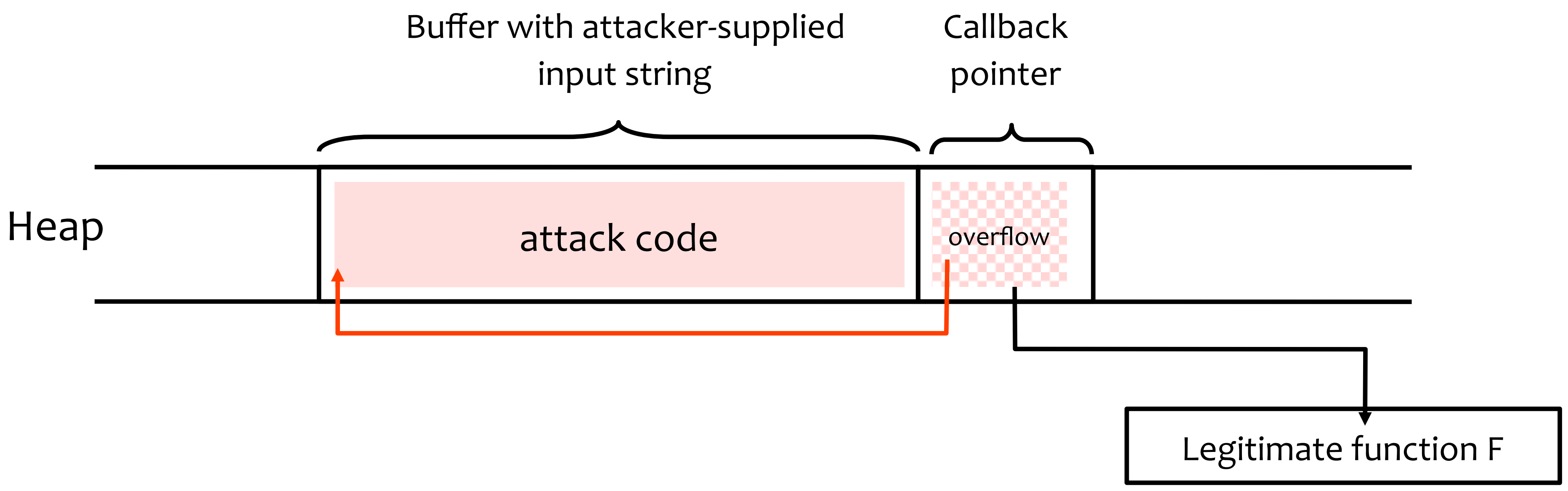
Defeating StackGuard

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



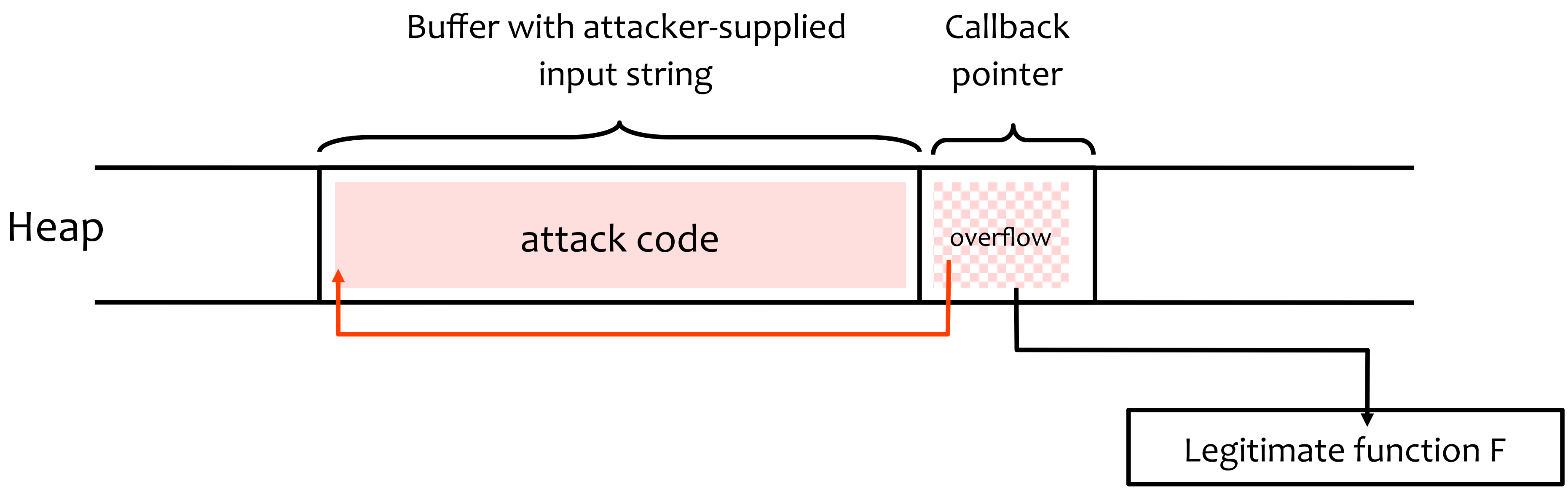
Function Pointer Overflow

- **Attack:** overflow a function pointer so that it points to attack code



Answer Q1

- **Attack:** overflow a function pointer so that it points to attack code

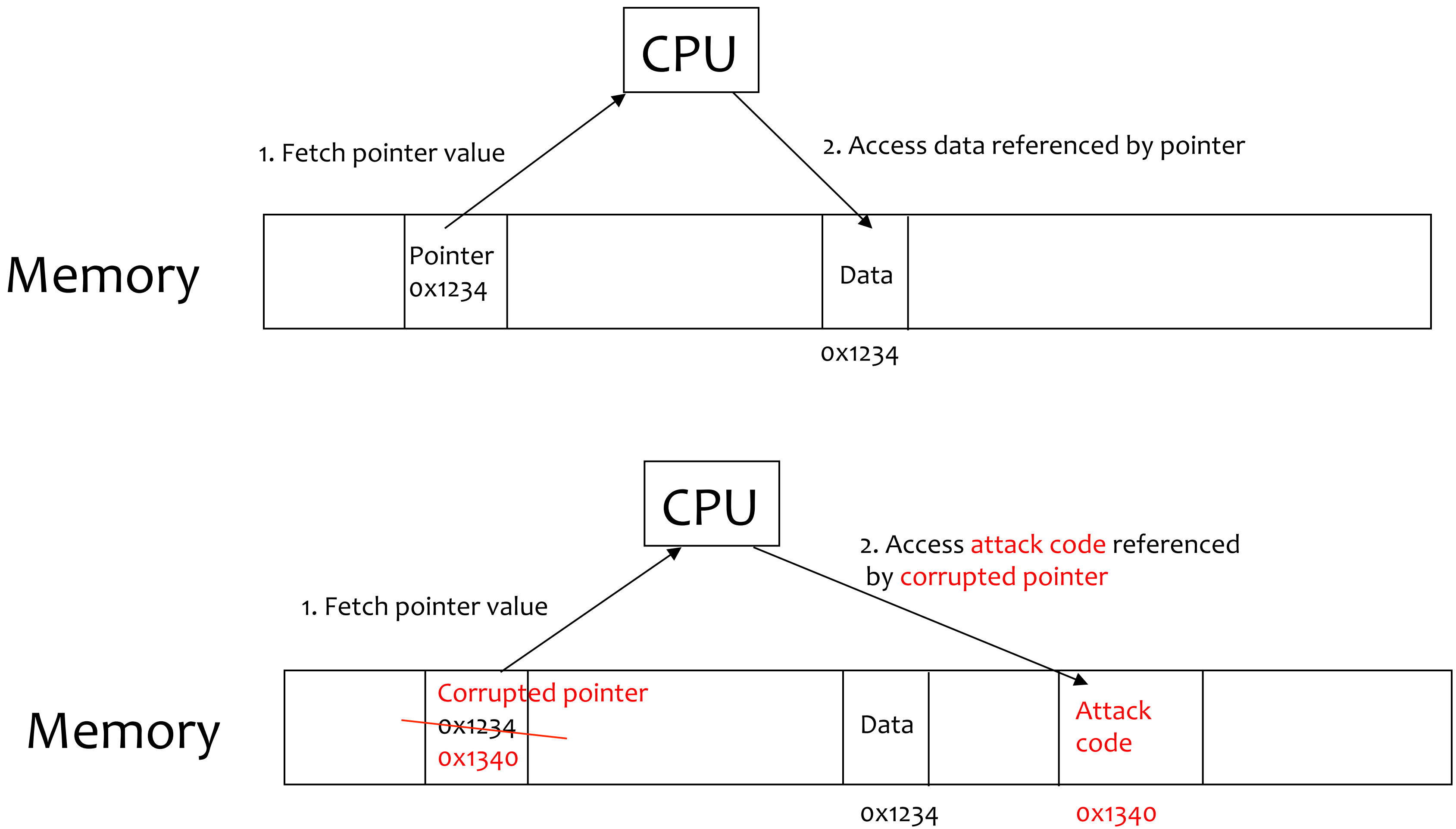


PointGuard

- Idea: encrypt all pointers while in memory
  - Generate a random key when program is executed
  - Each pointer is XORed with this key when loaded from memory to registers or stored back into memory
    - Pointers cannot be overflowed while in registers
- Attacker cannot predict the target program's key
  - Even if pointer is overwritten, after XORing with key it will dereference to a "random" memory address

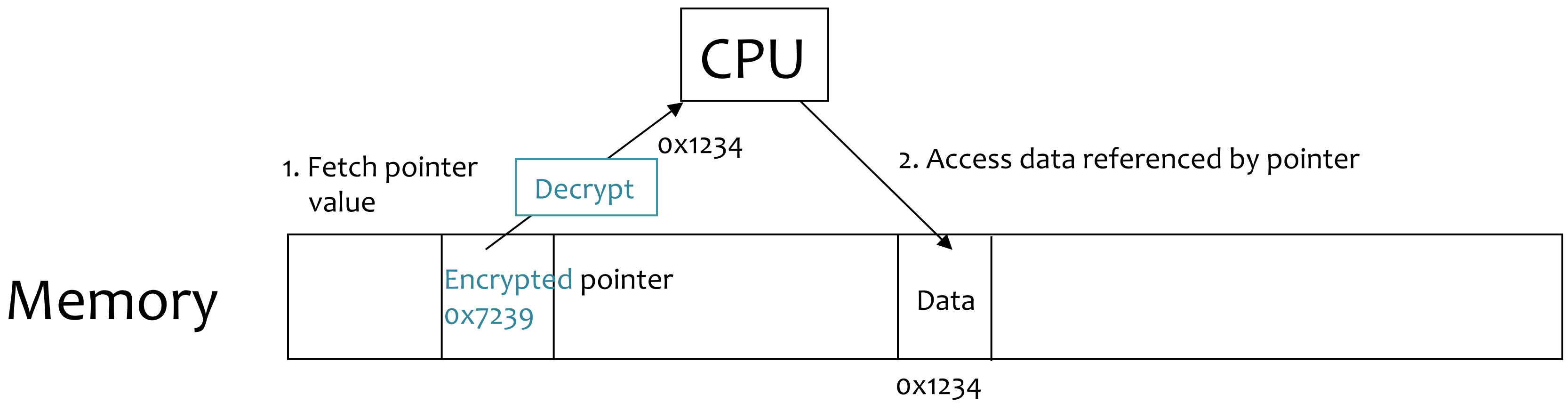
[Cowan]

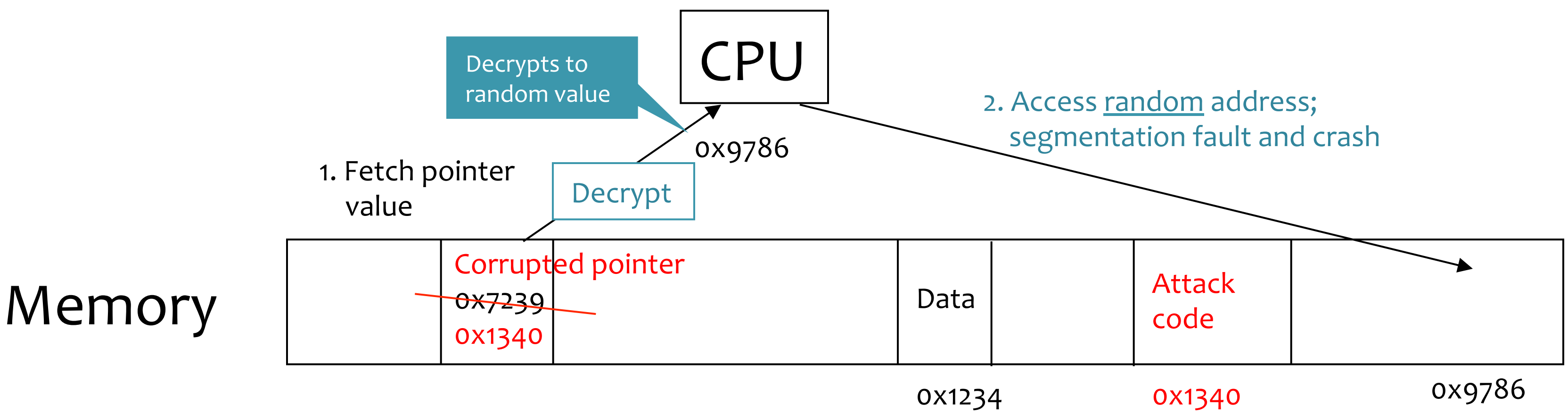
Normal Pointer Dereference



[Cowan]

PointGuard Dereference





PointGuard Issues

- Answer Q2

PointGuard Issues

- Must be very fast
  - Pointer dereferences are very common
- Compiler issues
  - Must encrypt and decrypt <u>only</u> pointers
  - If compiler "spills" registers, unencrypted pointer values end up in memory and can be overwritten there
- Attacker should not be able to modify the key
  - Store key in its own non-writable memory page
- PG'd code doesn't mix well with normal code
  - What if PG'd code needs to pass a pointer to OS kernel?

ASLR: Address Space Randomization

- Map shared libraries to a random location in process memory
  - Attacker does not know addresses of executable code
- Deployment (examples)
  - Windows Vista: 8 bits of randomness for DLLs
  - Linux (via PaX): 16 bits of randomness for libraries
  - Even Android
  - More effective on 64-bit architectures
- Other randomization methods
  - Randomize system call ids or instruction set

Example: ASLR in Vista

- Booting Vista twice loads libraries into different locations:

ntlanman.dll	0x6D7F0000	Microsoft® Lan Manager
ntmarta.dll	0x75370000	Windows NT MARTA provider
ntshrui.dll	0x6F2C0000	Shell extensions for sharing
ole32.dll	0x76160000	Microsoft OLE for Windows

ntlanman.dll	0x6DA90000	Microsoft® Lan Manager
ntmarta.dll	0x75660000	Windows NT MARTA provider
ntshrui.dll	0x6D9D0000	Shell extensions for sharing
ole32.dll	0x763C0000	Microsoft OLE for Windows

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

Other Possible Solutions

- Use safe programming languages, e.g., Java
  - What about legacy C code?
  - (Note that Java is not the complete solution)
- Static analysis of source code to find overflows
- Dynamic testing: "fuzzing"
- LibSafe: dynamically loaded library that intercepts calls to unsafe C functions and checks that there's enough space before doing copies
  - Also doesn't prevent everything

Even Modern Systems Don't Use These Defenses!

- Embedded systems
  - E.g., cars

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 they shouldn't)

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;
```

Integer Overflow and 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 and Implicit Cast

```
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/Side Channel 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

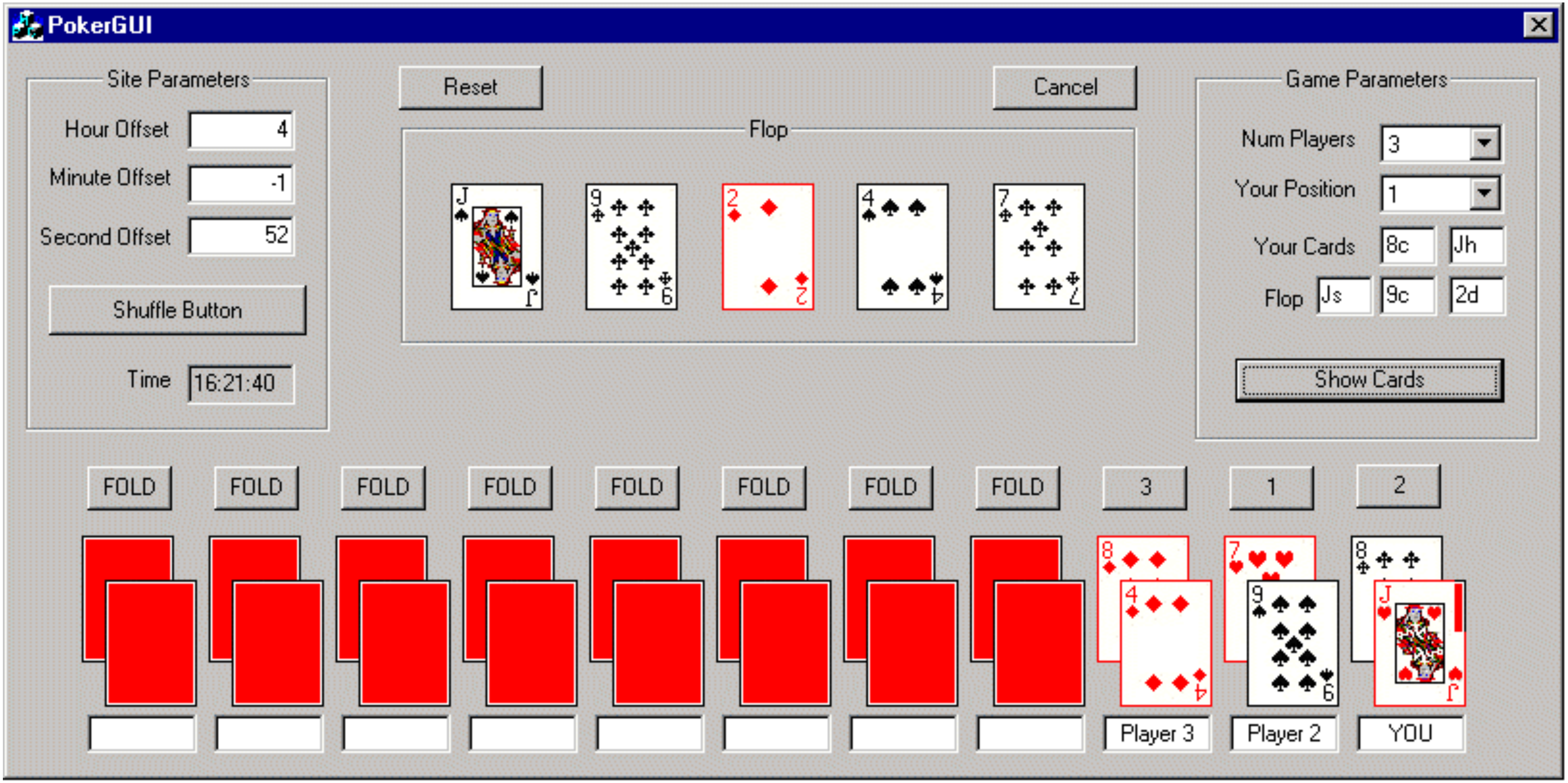
Other Side Channels

- David mentioned telescope + camera to read bits off **modem lights**
- Power usage
- Sound
- Error messages
- Facial expressions, tone of voice

Randomness Issues

- Many applications (especially security ones) require randomness
- Explicit uses:
  - Generate secret cryptographic keys
  - Generate random initialization vectors for encryption
- Other "non-obvious" uses:
  - Generate passwords for new users
  - Shuffle the order of votes (in an electronic voting machine)
  - Shuffle cards (for an online gambling site)


A World of Action!
Help
Suggestions
Fold in Turn
$0
jonnyboy
brute
Fold
Check
Bet ($5)
mamajoe
RickG
Options
Sit Out
Leave
584330
mamajoe: Hey guys, Big B is in!
Dealer Text



More details: "How We Learned to Cheat at Online Poker: A Study in Software Security"
http://www.cigital.com/papers/download/developer_gambling.php


A World of Action!
Help
Suggestions
Fold in Turn
$2
jonnyboy
brute
Fold
Show
mamajoe
RickG
Options
Sit Out
Leave
584330
mamajoe: Hey guys, Big B is in!
Dealer Text

PS3 and Randomness

Hackers obtain PS3 private cryptography key due to epic programming fail? (update)

http://www.engadget.com/2010/12/29/hackers-obtain-ps3-private-cryptography-key-due-to-epic-programm/

- 2010/2011: Hackers found/released private root key for Sony's PS3
- Key used to sign software – now can load any software on PS3 and it will execute as "trusted"
- Due to bad random number: same "random" value used to sign all system updates

PS3 and Randomness

- Example Current Event report from a past iteration of 484
  - https://catalyst.uw.edu/gopost/conversation/kohno/452868

Posted Jan 3, 2011 5:01 PM Quote

PS3 Exploit
Today, January 3rd, George "Geohot" Hotz found and released the private root key for Sony's Playstation 3 (PS3) video game console (http://www.geohot.com/). What this means is that homebrew software enthusiasts, scientists, and software pirates can now load arbitrary software on the PS3 and sign it using this key, and the system will execute it as trusted code. Legitimately, this allows Linux and other operating systems to take advantage of the PS3's cell processor architecture; however, it also opens up avenues of software piracy previously impossible on Sony's system without requiring any hardware modifications to the system (previous access of this kind required a USB hardware dongle)

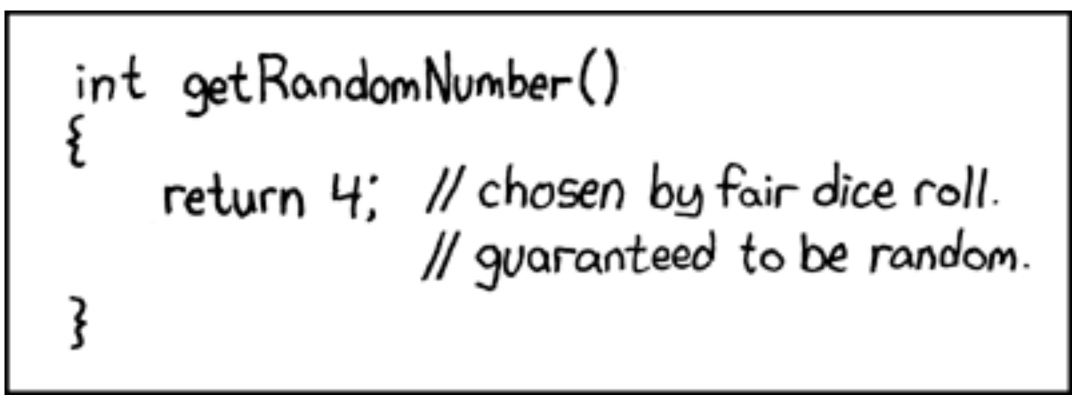
How it Was Done
This was enabled by a cryptographic error by Sony developers in their update process. In the DSA signature algorithm, a number k is chosen from a supposedly random source for each signed message. So long as the numbers are unique, the system is secure, but duplicating a random number between messages can expose the private key to an untrusted party using simple mathematics (http://rdist.root.org/2010/11/19/dsa-requirements-for-random-k-value/). Sony used the exact same "random value" k for all updates pushed to the system, making the signature scheme worthless.

The Most Secure
After Sony removed the "other OS" functionality of the PS3, greater scrutiny was placed on the PS3. Since it's release in 2006, the Playstation 3 was considered the most secure of the three major video game consoles, as it was the only console without a "root" compromise in the four years since release (there were vulnerabilities limited to specific firmware or that required specialized hardware, but nothing that provided unfettered access). By comparison, Microsoft's Xbox 360 was cracked over 4 years ago (http://www.theregister.co.uk/2007/03/01/xbox_hack), and the Wii was cracked over 2 years ago (http://wiibrew.org/wiki/Index.php).

Cullen Walsh
Mark Jordan
Peter Lipay

Other Problems

- Key generation
  - Debian removed the randomness from SSL, creating vulnerable keys for thousands of users/servers
  - Undetected for 2 years (2006-2008)
- Live CDs, diskless clients
  - May boot up in same state every time
- Virtual Machines
  - Save state: Opportunity for attacker to inspect the pseudorandom number generator's state
  - Restart: May use same "psuedorandom" value more than once



https://xkcd.com/221/

Obtaining Pseudorandom Numbers

- For security applications, want "cryptographically secure pseudorandom numbers"
- Libraries include cryptographically secure pseudorandom number generators
- Linux:
  - /dev/random
  - /dev/urandom - nonblocking, possibly less entropy
- Internally:
  - Entropy pool gathered from multiple sources

Where do (good) random numbers come from?

- Humans: keyboard, mouse input
- Timing: interrupt firing, arrival of packets on the network interface
- Physical processes: unpredictable physical phenomena