

CSE 484 / CSE M 584: Computer Security and Privacy

Software Security: Misc and Principles

Spring 2015

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

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

Looking Forward

- Today: one more day on software security
- Friday: guest lecture by David Aucsmith
- Next week: start crypto

- Ethics form all done – yay!
- Homework #1 due April 17
- Lab #1 out TODAY
 - Please send a group name and a public key via email (pasted, NOT attached) to Peter <neyp@cs.washington.edu>
 - Instructions for creating a key are in the lab description

- Section this week: Lab 1

Another Type of Vulnerability

- Consider this code:

```
int openfile(char *path) {
    struct stat s;
    if (stat(path, &s) < 0)
        return -1;
    if (!S_ISREG(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_ISREG(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)

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 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

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)

C's rand() Function

- C has a built-in random function: `rand()`

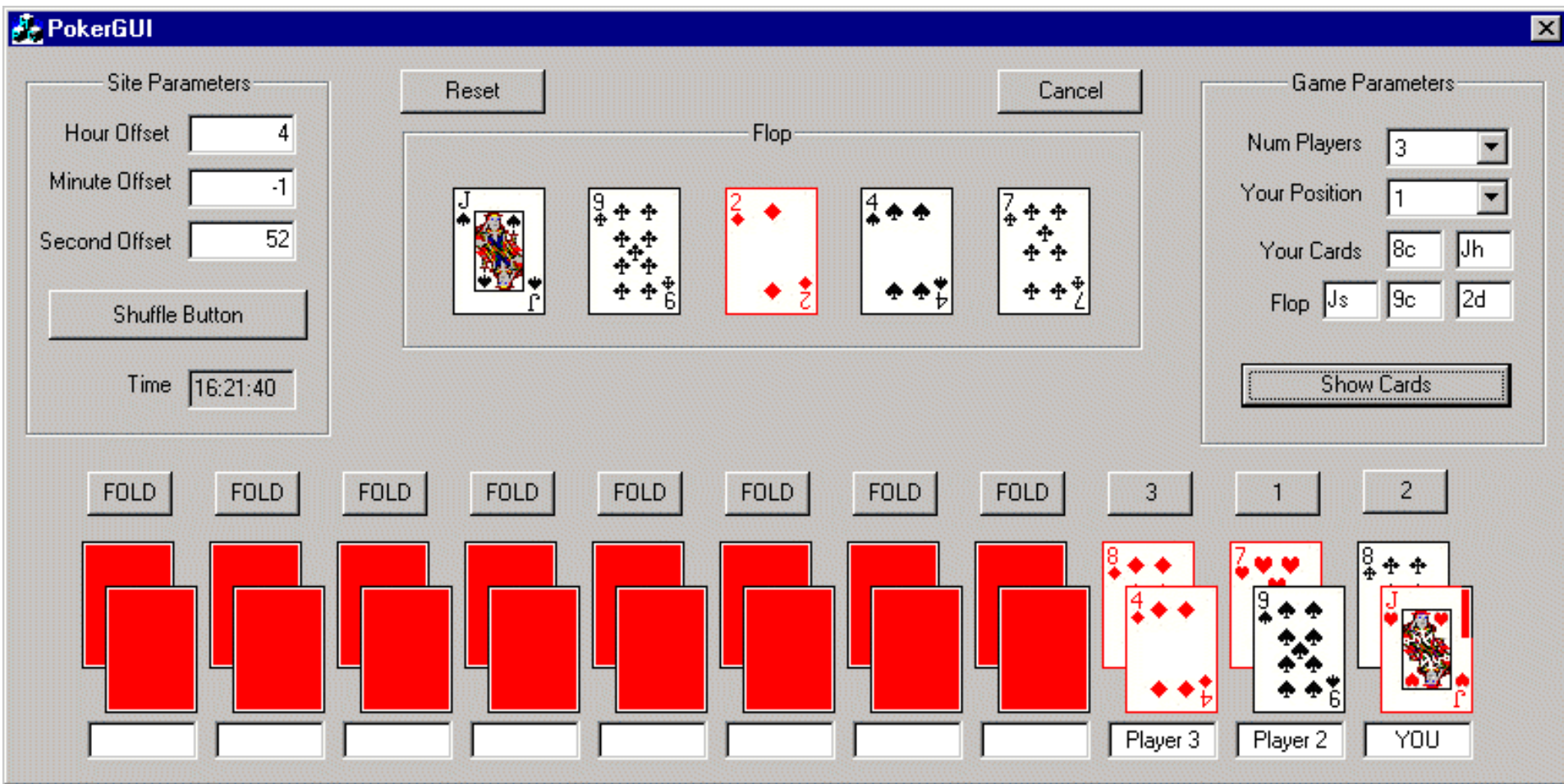
```
unsigned long int next = 1;
/* rand:  return pseudo-random integer on 0..32767 */
int rand(void) {
    next = next * 1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}
/* srand:  set seed for rand() */
void srand(unsigned int seed) {
    next = seed;
}
```

- Problem: don't use `rand()` for security-critical applications!
 - Given a few sample outputs, you can predict subsequent ones

Problems in Practice

- One institution used (something like) `rand()` to generate passwords for new users
 - Given your password, you could predict the passwords of other users
- Kerberos (1988 - 1996)
 - Random number generator improperly seeded
 - Possible to trivially break into machines that rely upon Kerberos for authentication
- Online gambling websites
 - Random numbers to shuffle cards
 - Real money at stake
 - But what if poor choice of random numbers?





More details: “How We Learned to Cheat at Online Poker: A Study in Software Security”

http://www.cigital.com/papers/download/developer_gambling.php

PS3 and Randomness

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

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 its 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>).

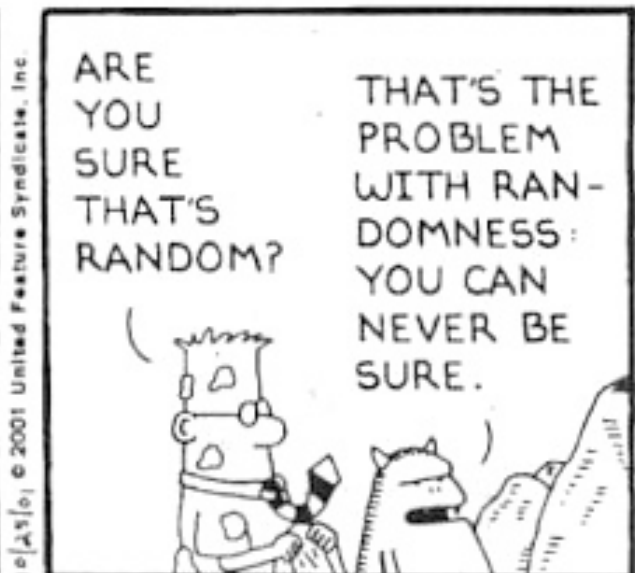
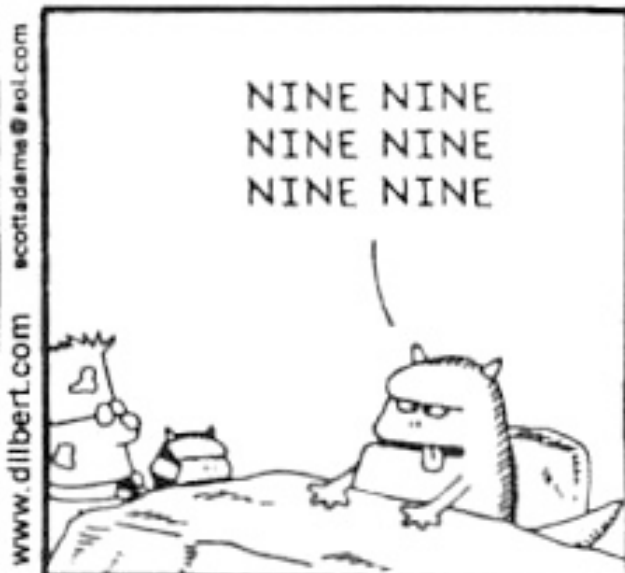
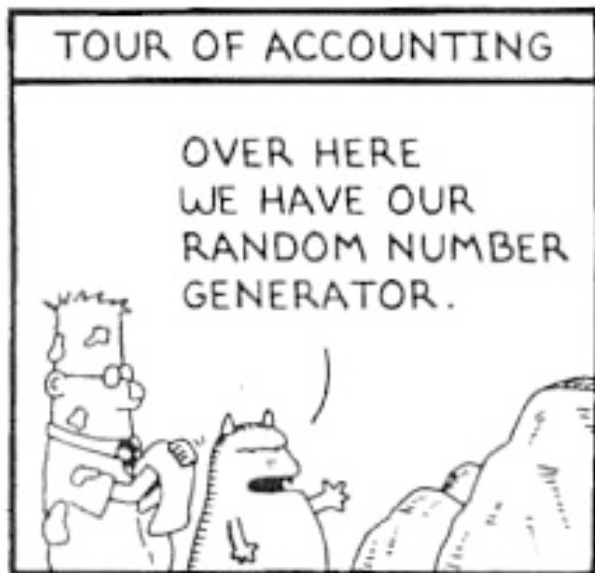
Cullen Walsh

Mark Jordan

Peter Lipay

Other Problems

- Key generation
 - Ubuntu 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



```
int getRandomNumber()  
{  
    return 4; // chosen by fair dice roll.  
              // guaranteed to be random.  
}
```

<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

Software Security: So what do we do?

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

General Principles

- Check inputs

Shellshock

- Check inputs: not just to prevent buffer overflows
- **Example: Shellshock (September 2014)**
 - Vulnerable servers processed input from web requests, passed (user-provided) environment variables (like user agent, cookies...) to CGI scripts
 - Maliciously crafted environment variables exploited a bug in bash to execute arbitrary code

```
env x='() { :; }; echo OOPS' bash -c :
```

General Principles

- Check inputs
- Check all return values
- Least privilege
- Securely clear memory (passwords, keys, etc.)
- Failsafe defaults
- Defense in depth
 - Also: prevent, detect, respond
- NOT: security through obscurity

General Principles

- Reduce size of trusted computing base (TCB)
- Simplicity, modularity
 - But: Be careful at interface boundaries!
- Minimize attack surface
- Use vetted component
- Security by design
 - But: tension between security and other goals
- Open design? Open source? Closed source?
 - Different perspectives