Software Security: Attacks, Defenses, and Design Principles

Tadayoshi Kohno

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

- Randomness
- Timing Attacks
- Defensive Approaches
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()`

```c
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.
Randomness and the Netscape Browser

How secure is the World Wide Web?

Ian Goldberg and David Wagner

No one was more surprised than Netscape Communications when a pair of computer-science students broke the Netscape encryption scheme. Ian and David describe how they attacked the popular Web browser and what they found out.
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
  - 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?
Big news... CNN, etc..
Other Problems

◆ 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
Obtaining Pseudorandom Numbers

- For security applications, want “cryptographically secure pseudorandom numbers”

- Libraries include:
  - OpenSSL
  - Microsoft’s Crypto API

- Linux:
  - /dev/random
  - /dev/urandom

- Internally:
  - Pool from multiple sources (interrupt timers, keyboard, ...)
  - Physical sources (radioactive decay, ...)
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
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)

```plaintext
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
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: $256^8 = 2048$
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
Toward Preventing Buffer Overflow

- Use safe programming languages, e.g., Java
  - What about legacy C code?
- Static analysis of source code to find overflows
- Black-box testing with long strings
- Mark stack as non-executable
- Randomize stack location or encrypt return address on stack by XORing with random string
  - Attacker won’t know what address to use in his or her string
- Run-time checking of array and buffer bounds
  - StackGuard, libsafe, many other tools
Non-Executable Stack

- NX bit for pages in memory
  - Modern Intel and AMD processors support
  - Modern OS support as well
- Some applications need executable stack
  - For example, LISP interpreters
- Does not defend against return-to-libc exploits
  - Overwrite return address with the address of an existing library function (can still be harmful)
- ...nor against heap and function pointer overflows
- ...nor changing stack internal variables (auth flag, ...
Run-Time Checking: StackGuard

- Embed “canaries” 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
- PointGuard also places canaries next to function pointers and setjmp buffers
  - Worse performance penalty
- StackGuard doesn’t completely solve the problem (can be defeated)
Defeating StackGuard (Sketch)

Idea: overwrite pointer used by some `strcpy` and make it point to return address (RET) on stack

- `strcpy` will write into RET without touching canary!

Suppose program contains `strcpy(dst,buf)`
PointGuard

- Attack: overflow a function pointer so that it points to attack code
- 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 overflown 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
Normal Pointer Dereference  [Cowan]

1. Fetch pointer value
2. Access data referenced by pointer

Memory

CPU

1. Fetch pointer value
2. Access attack code referenced by corrupted pointer

Memory
PointGuard Dereference

1. Fetch pointer value

2. Access data referenced by pointer

Memory

CPU

Decrypted
0x1234

Data

0x1234

Corrupted pointer
0x7239
0x1340

1. Fetch pointer value

2. Access random address; segmentation fault and crash

Attack
code

0x1340

0x9786

0x1234

0x9786

0x7239

0x1234

0x1340

[Source: Cowan]
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