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 (bit more)
- Timing Attacks
- Defensive Approaches
Tour of Accounting

Over here we have our random number generator.

Nine nine nine nine nine nine nine nine.

Are you sure that's random?

That's the problem with randomness: you can never be sure.
IN THE RUSH TO CLEAN UP THE DEBIAN-OPENSSL FIASCO, A NUMBER OF OTHER MAJOR SECURITY HOLES HAVE BEEN UNCOVERED:

<table>
<thead>
<tr>
<th>AFFECTED SYSTEM</th>
<th>SECURITY PROBLEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>FEDORA CORE</td>
<td>VULNERABLE TO CERTAIN DECODER RINGS</td>
</tr>
<tr>
<td>XANDROS (EEE PC)</td>
<td>GIVES ROOT ACCESS IF ASKED IN STERN VOICE</td>
</tr>
<tr>
<td>GENTOO</td>
<td>VULNERABLE TO FLATTERY</td>
</tr>
<tr>
<td>OLPC OS</td>
<td>VULNERABLE TO JEFF GOLDBLUM’S POWERBOOK</td>
</tr>
<tr>
<td>SLACKWARE</td>
<td>GIVES ROOT ACCESS IF USER SAYS ELVISH WORD FOR “FRIEND”</td>
</tr>
<tr>
<td>UBUNTU</td>
<td>TURNS OUT DISTRO IS ACTUALLY JUST WINDOWS VISTA, WITH A FEW CUSTOM THEMES</td>
</tr>
</tbody>
</table>
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
  • $\text{PwdCheck}(\text{RealPwd}, \text{CandidatePwd})$ should:
    – Return TRUE if RealPwd matches CandidatePwd
    – Return FALSE otherwise
  • RealPwd and CandidatePwd are both 8 characters long

❖ Implementation (like TENEX system)

$\text{PwdCheck}(\text{RealPwd}, \text{CandidatePwd})$  // both 8 chars
  for $i = 1$ to 8 do
    if ($\text{RealPwd}[i] \neq \text{CandidatePwd}[i]$) then
      return FALSE
  return TRUE

❖ Clearly meets functional description
**Attacker Model**

PwCheck(RealPwd, CandidatePwd) // both 8 chars

for $i = 1$ to 8 do

    if (RealPwd[$i$] $\neq$ 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:** $256 \times 8 = 2048$
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 and signature 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
- 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, ProPolice, 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)
  - Newer: Return-oriented programming
- ...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 (Example, 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)`

- Overwrite destination of strcpy with RET position
- strcpy will copy BadPointer here
- Return execution to this address
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

Pointer 0x1234

Data

0x1234

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

Memory

CPU

Corrupted pointer 0x1234

Data

Attack code

0x1234 0x1340
PointGuard Dereference

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

Memory
- CPU
  - Decrypt
  - 0x1234

0x7239
0x1234

Encrypt

Corrupted pointer
- 0x7239
- 0x1340

0x1340

Data

2. Access random address; segmentation fault and crash

CPU
- Decrypt
- Decrypts to random value
- 0x9786

Memory
- CPU
  - 0x9786

0x9786

Data

Attack code
PointGuard Issues

- Must be very fast
  - Pointer dereferences are very common
- Compiler issues
  - Must encrypt and decrypt only 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?
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
Genetic Diversity

- Problems with Monoculture

- Steps toward diversity
  - Automatic diversification of compiled code
  - Address Space Randomization
Principles

- Open design? Open source?
- Maybe...
