
Daniel Halperin
Tadayoshi Kohno

Thanks to Dan Boneh, Dieter Gollmann, John Manferdelli, John Mitchell, Vitaly Shmatikov, Bennet Yee, and many others for sample slides and materials ...
Updates Oct. 7th

• Coffee/tea signup sheet posted (optional)
• M 584 reading for Oct. 14th posted
• Security reviews & Current events
• Lab 1
Today

• Randomness
• Software defenses
• Advanced attacks
• Advanced defense
TOUR OF ACCOUNTING

OVER HERE WE HAVE OUR RANDOM NUMBER GENERATOR.

NINE NINE NINE NINE NINE

ARE YOU SURE THAT'S RANDOM?

THAT'S THE PROBLEM WITH RANDOMNESS: YOU CAN NEVER BE SURE.
How would you test a RNG?
How would you test a RNG?

- **Statistical tests**: how are the output values distributed?
- **Spectral tests**: plot data in n-D, find patterns
- Related to compressibility/summarizibility
  
  A: 010101010101010101010101010101
  
  B: 110010000110001110111101101010
RANDU - famously bad PRNG

- $X[i+1] = 65539 \times X[i] \pmod{2^{32}}$
- All $X[i]$ are odd!

3-D plot of RANDU output
(Wikipedia, RANDU article)
RANDU - famously bad PRNG

One of us recalls producing a “random” plot with only 11 planes, and being told by his computer center’s programming consultant that he had misused the random number generator: “We guarantee that each number is random individually, but we don’t guarantee that more than one of them is random.” Figure that out.


(Wikipedia, RANDU article)
Where do (good) random numbers come from?
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
SGI’s LavaRand

(https://hackaday.com/2005/06/05/lava-lamp-random-number-generator/)
Open Source LavaRnd

- Camera CCD looking into an empty, dark, shielded can
- Measuring background radiation “thermal noise”
- Quantum process: randomness from Heisenberg’s Uncertain Principle

(http://www.lavarnd.org/what/process.html)
Physical RNGs in CPUs

- **State of uninitialized memory** when machine powers on

- Tiny *variations in voltage* over resistor

(Holcomb, Burleson, Fu, IEEE Trans. Comp 58(9), Sept. 2009)
Obtaining Pseudorandom Numbers

- For security applications, want “cryptographically secure pseudorandom numbers”
- Libraries include:
  - OpenSSL
  - Microsoft’s Crypto API
- Linux:
  - /dev/random
  - /dev/urandom - nonblocking, possibly less entropy
- Internally:
  - Entropy pool gathered from multiple sources
  - Physical sources
void foo (char *argv[]) {
  push  %ebp
  mov   %esp,%ebp
  char buf[128];
  sub    $0x88,%esp
  mov    0x8(%ebp),%eax
  strcpy(buf, argv[1]);
  add    $0x4,%eax
  mov    (%eax),%eax
  mov    %eax,0x4(%esp)
  lea    -0x80(%ebp),%eax
  mov    %eax,(%esp)
  call   804838c <strcpy@plt>
}
leave
ret
void foo (char *argv[]) {
  push %ebp
  mov %esp,%ebp
  char buf[128];
  sub $0x88,%esp
  mov 0x8(%ebp),%eax
  strcpy(buf, argv[1]);
  add $0x4,%eax
  mov (%eax),%eax
  mov %eax,0x4(%esp)
  lea -0x80(%ebp),%eax
  mov %eax,(%esp)
  call 804838c <strcpy@plt>
}
void foo (char *argv[]) {
    int canary = <random>;
    char buf[128];
    strcpy(buf, argv[1]);
    assert(canary unchanged);
}
void foo (char *argv[])
{
    int canary = <random>;
    char buf[128];
    strcpy(buf, argv[1]);
    assert(canary unchanged);
}

Stack Canary

Any Canary Advice?
void foo (char *argv[]) {
    int canary = <random>;
    char buf[128];
    strcpy(buf, argv[1]);
    assert(canary unchanged);
}

Any Canary Advice?
• Null byte stops strcpy() bugs
• CR-LF stops gets() bugs
• EOF stops fread() bugs
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)`
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)`
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)`

Return execution to this address
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

Return execution to this address
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
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, ...)

Monday, October 10, 11
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 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
Normal Pointer Dereference

[1] Fetch pointer value

Memory

CPU

Corrupted pointer
0x1234
0x1340

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

CPU

Memory

Corrupted pointer
0x1234
0x1340

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

Monday, October 10, 11
PointGuard Dereference

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

1. Fetch pointer value
2. Access random address; segmentation fault and crash

Monday, October 10, 11
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?
Other solutions

- Use safe programming languages, e.g., Java
  - What about legacy C code?
- **Static analysis** of source code to find overflows
- 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
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)

  ```
  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

\[
\text{PwdCheck}(\text{RealPwd}, \text{CandidatePwd}) \quad // \text{both 8 chars}
\]

\[
\text{for } i = 1 \text{ to } 8 \text{ do}
\]

\[
\text{if } (\text{RealPwd}[i] \neq \text{CandidatePwd}[i]) \text{ then}
\]

\[
\text{return FALSE}
\]

\[
\text{return TRUE}
\]

\[\blacktriangleleft \text{Attacker can guess } \text{CandidatePwds} \text{ through some standard interface} \]

\[\blacktriangleleft \text{Naive: Try all } 256^8 = 18,446,744,073,709,551,616 \text{ possibilities} \]
Attacker Model

PwdCheck(RealPwd, CandidatePwd) // both 8 chars

for i = 1 to 8 do

  sleep for 1 second
  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
        sleep for 1 second
        if (RealPwd[i] != CandidatePwd[i]) then
            return FALSE
    return TRUE
```

- 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