CSE 484 / CSE M 584 (Spring 2012)

Software Security: Buffer Overflow Attacks and Beyond

Tadayoshi Kohno

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

Administrivia

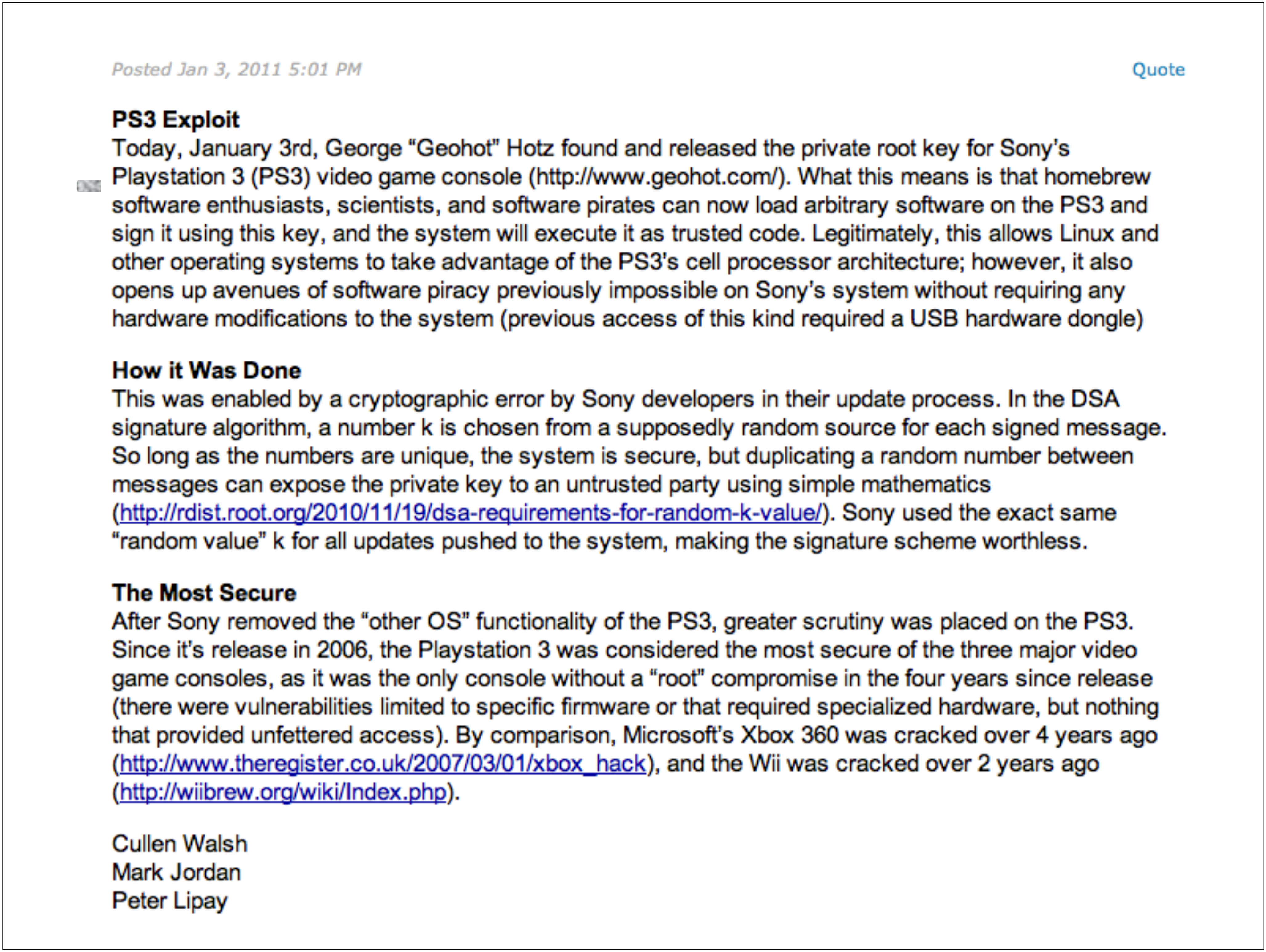
- Coffee/teas -- meet in CSE atrium
- Lab 1 out

Goals for Today

- Software security
    - Continue

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

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.

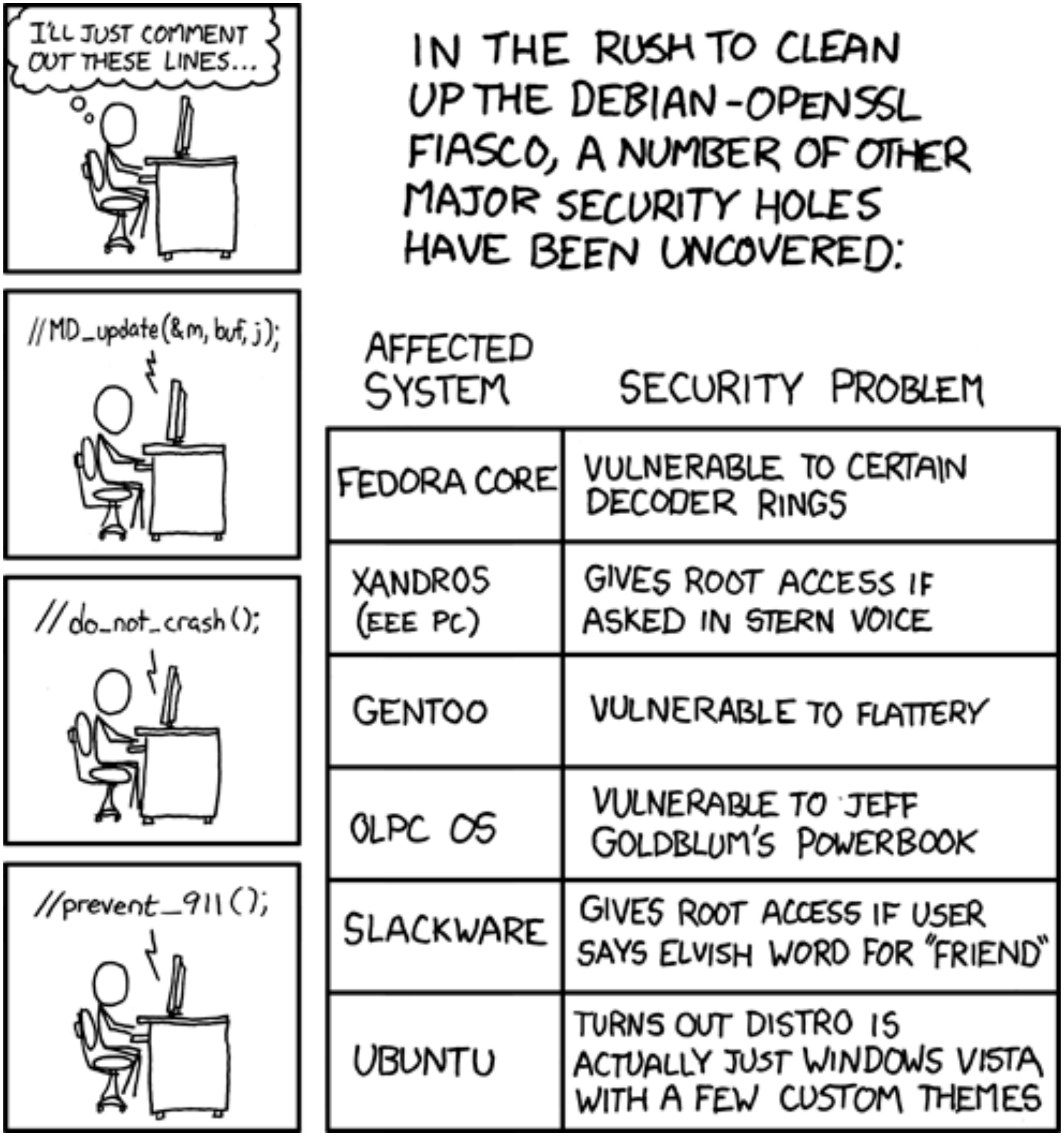
The Most Secure

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
  - 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



IN THE RUSH TO CLEAN UP THE DEBIAN-OPENSSL FIASCO, A NUMBER OF OTHER MAJOR SECURITY HOLES HAVE BEEN UNCOVERED:

AFFECTED SYSTEM	SECURITY PROBLEM
FEDORA CORE	VULNERABLE TO CERTAIN DECODER RINGS
XANDROS (EEE PC)	GIVES ROOT ACCESS IF ASKED IN STERN VOICE
GENTOO	VULNERABLE TO FLATTERY
OLPC OS	VULNERABLE TO JEFF GOLDBLUM'S POWERBOOK
SLACKWARE	GIVES ROOT ACCESS IF USER SAYS ELVISH WORD FOR "FRIEND"
UBUNTU	TURNS OUT DISTRO IS ACTUALLY JUST WINDOWS VISTA WITH A FEW CUSTOM THEMES

Source: XKCD


DILBERT By Scott Adams
TOUR OF ACCOUNTING
OVER HERE WE HAVE OUR RANDOM NUMBER GENERATOR.
scottadams@aol.com
www.dilbert.com
NINE NINE NINE NINE NINE NINE
10/25/01 © 2001 United Feature Syndicate, Inc.
ARE YOU SURE THAT'S RANDOM?
THAT'S THE PROBLEM WITH RANDOMNESS: YOU CAN NEVER BE SURE.

Obtaining Pseudorandom Numbers

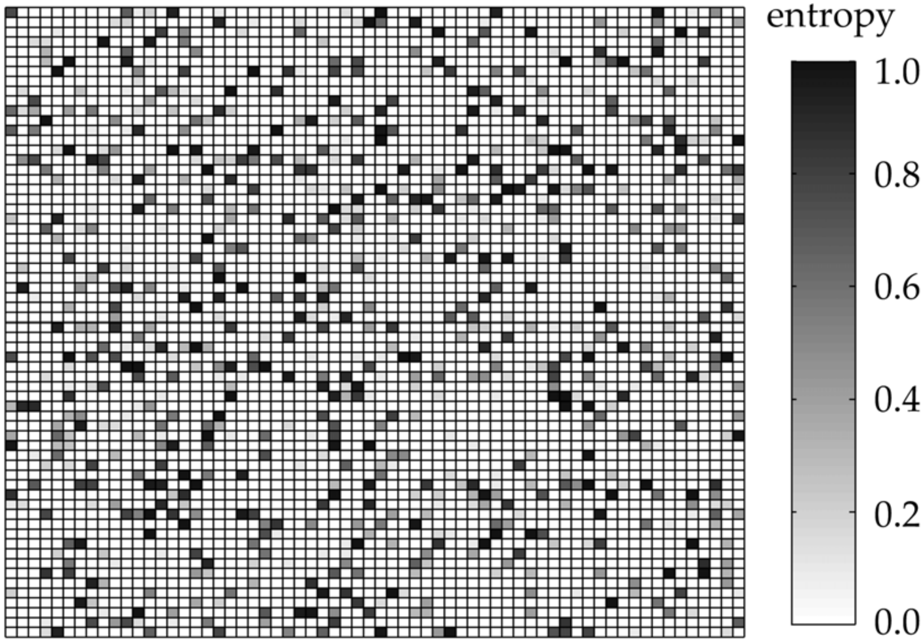
- For security applications, want "cryptographically secure pseudorandom numbers"
- Libraries include cryptographically secure pseudorandom number generators
- Linux:
  - /dev/random
  - /dev/urandom - nonblocking, possibly less entropy
- Internally:
  - Entropy pool gathered from multiple sources

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

Physical RNGs in CPUs

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

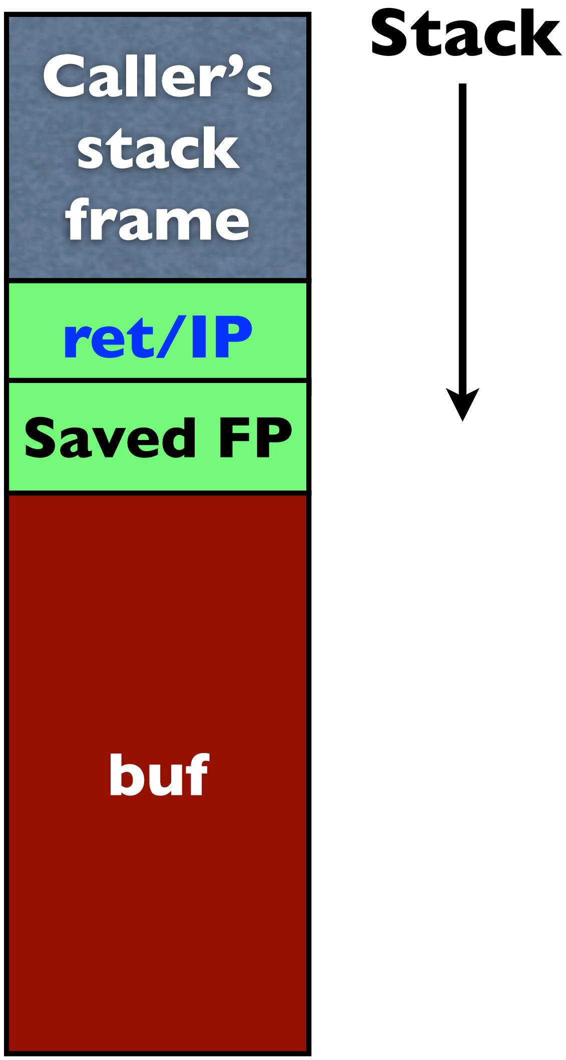


(Holcomb, Burleson, Fu, IEEE Trans. Comp 58(9), Sept. 2009)

- Tiny ***variations in voltage*** over resistor

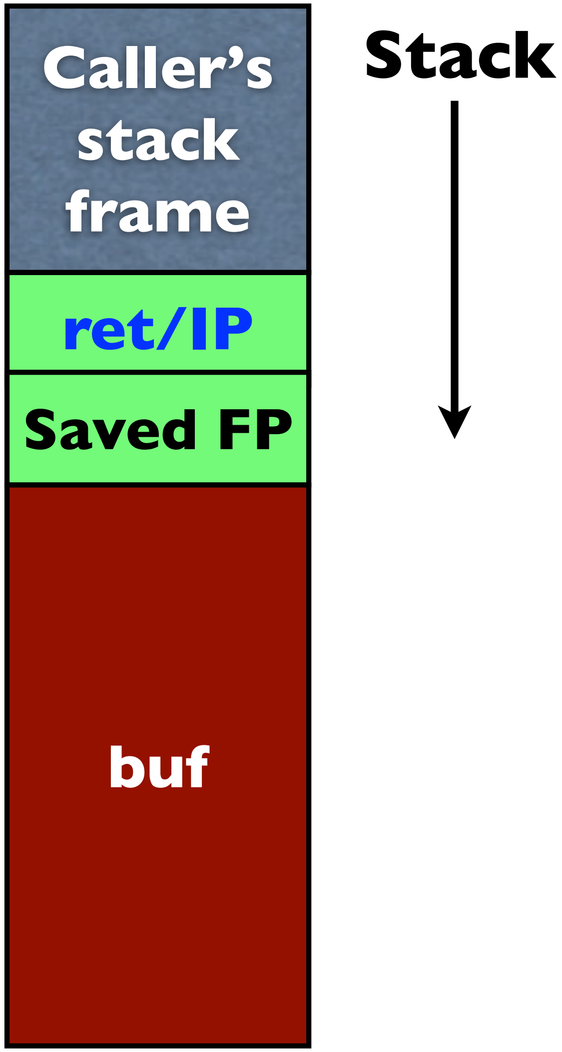
Buffer overflow attacks

```
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
```



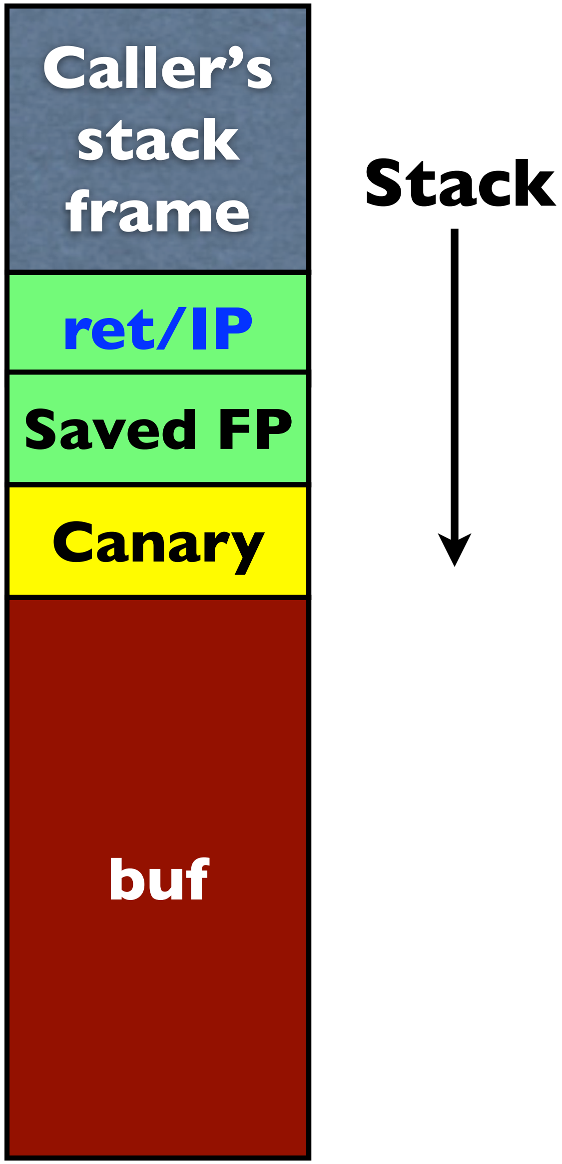
How to defend against this?

```
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
```



Stack Canary (StackGuard)

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



Any Canary Advice?

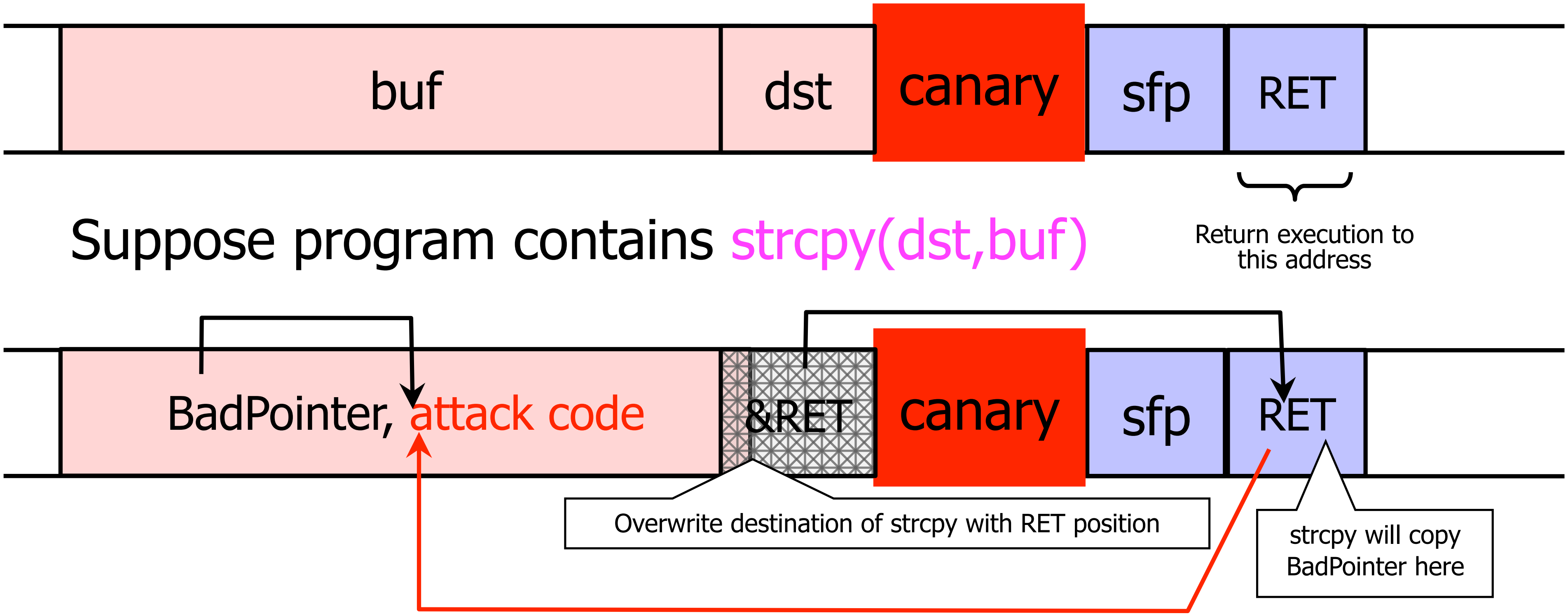
- Null byte stops strcpy() bugs
- CR-LF stops gets() bugs
- EOF stops fread() bugs

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!



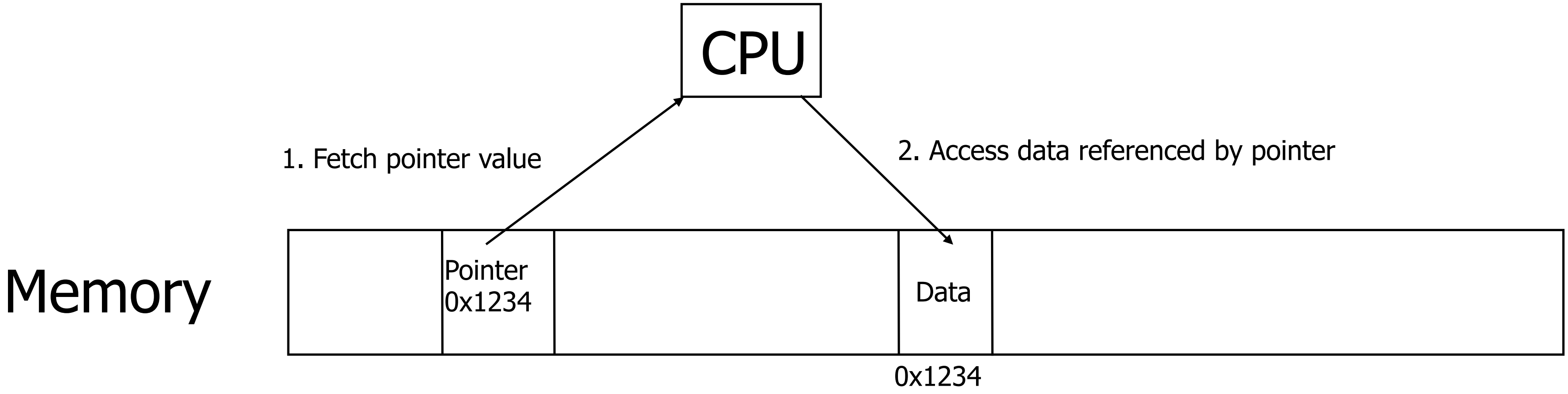
Non-Executable Stack

- 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)
  - Generalization: Return-oriented programming
- …nor against heap and function pointer overflows
- …nor changing stack internal variables (auth flag, …)

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]



CPU

1. Fetch pointer value

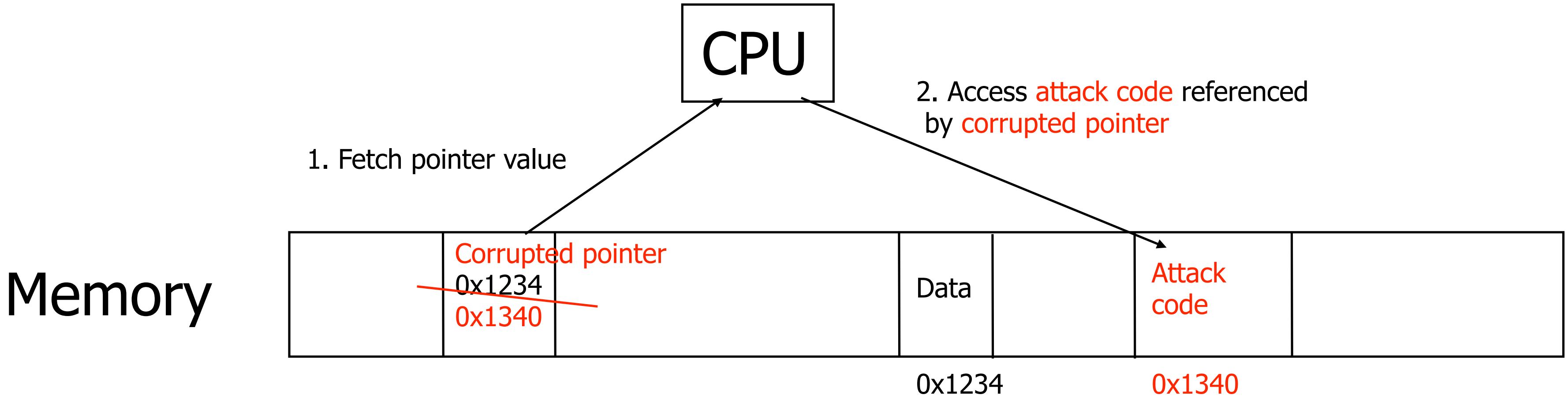
2. Access data referenced by pointer

Memory

Pointer
0x1234

Data

0x1234



CPU

1. Fetch pointer value

2. Access attack code referenced
by corrupted pointer

Memory

Corrupted pointer
~~0x1234~~
0x1340

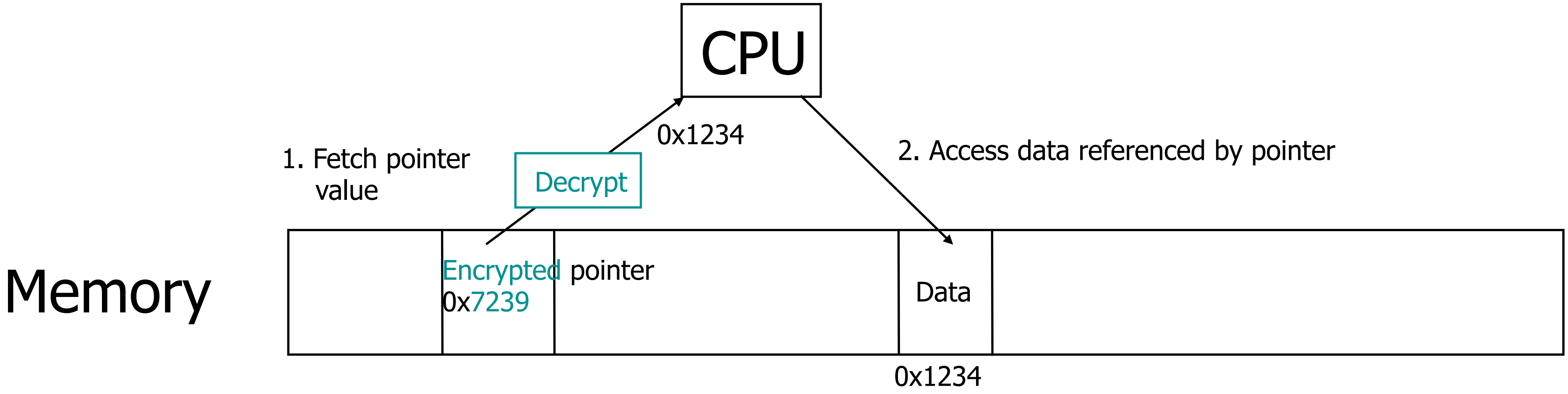
Data

Attack
code

0x1234

0x1340

PointGuard Dereference [Cowan]



CPU

0x1234

1. Fetch pointer value

Decrypt

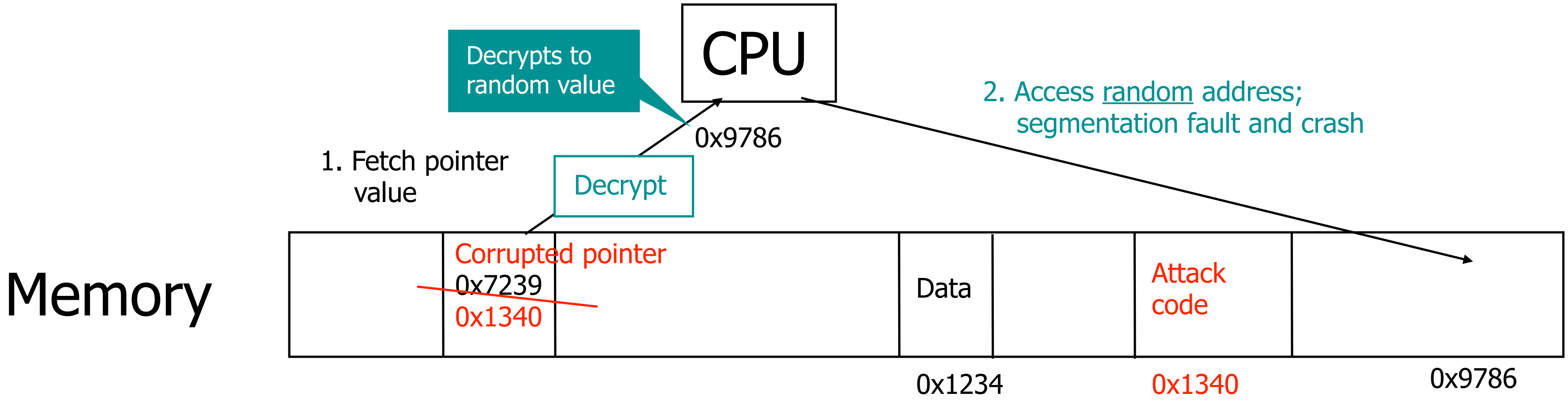
2. Access data referenced by pointer

Memory

Encrypted pointer
0x7239

Data

0x1234



Decrypts to random value

CPU

0x9786

2. Access random address; segmentation fault and crash

1. Fetch pointer value

Decrypt

Memory

Corrupted pointer
~~0x7239~~
0x1340

Data

Attack code

0x1234

0x1340

0x9786

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 to some of these issues

- 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
- 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

```
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*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

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

- Example in Tor:
  - users get lists of relays from "directory authorities"
  - require signatures from 4/7 authorities to accept
  - variety of OS'es, crypto libs, etc.
  - Works: only 3 servers compromised by Debian SSL bug

Principles

- Open design? Open source?
- Maybe…

- Linux Kernel Backdoor Attempt: http://www.freedom-to-tinker.com/?p=472

- PGP Corporation: http://www.symantec.com/connect/downloads/symantec-pgp-desktop-peer-review-source-code