Memory Exploits & Defenses

Presenter: Kevin Snow

What is the threat?

How do we defend ourselves?

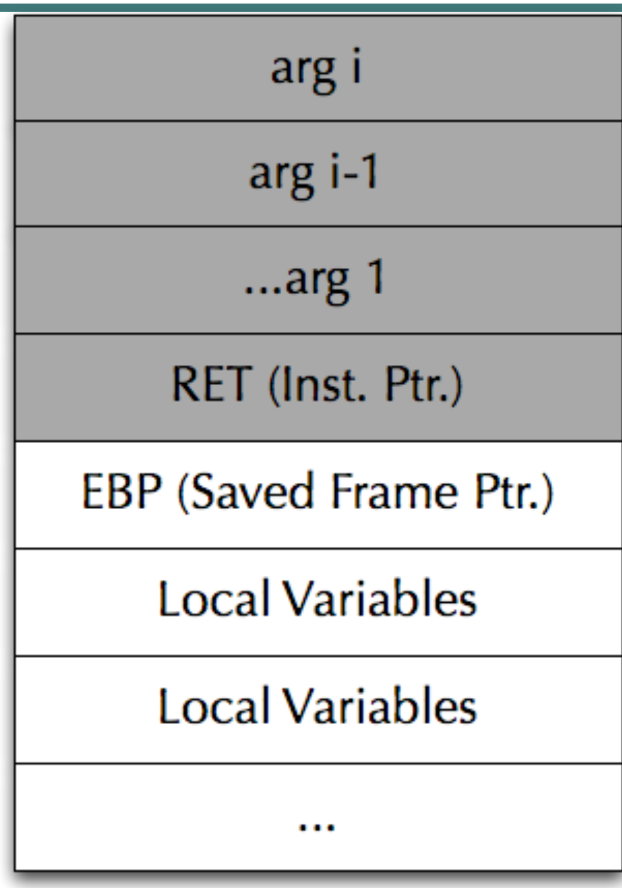
What is the threat?

- Stack Smashing
- Return-to-libc
- Format String Error
- Heap Overflow

Generic Stack Frame

Caller

Callee



Stack Smashing

Goal:

Point return address to our buffer, which contains executable code

Stack Smashing

```
void f(char *p){
  char x[128];
  strcpy(x, p);
}
```

p
\x30\xfb\xff\xbf
\xeb\x2a\x5e\x89
\xeb\x2a\x5e\x89
\x90\x90\x90\x90
\x90\x90\x90\x90
\x90\x90\x90\x90

Our Stack

arg 1
RET
EBP
Local Variables
Local Variables
...
Local Variables

Generic Stack

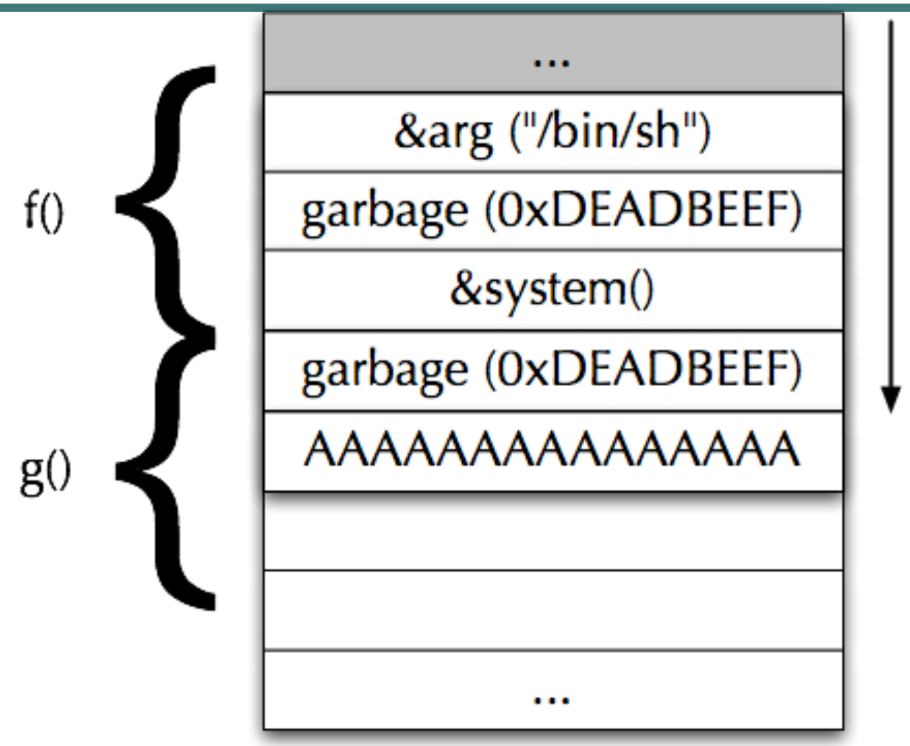
return-to-libc

Goal:
Point return address to an existing library function

return-to-libc

```
f(){
    g(&foo);
}

g(char *x){
    char y [SZ];
    scanf(y);
}
```



Linked libraries often have useful strings lying around

Format String Errors

- Goal: Take advantage of *printf()* family of functions

Good:
```
printf("%d", num);
```

Bad:
```
printf("%d");
```

Good:
```
printf("%s", myString);
```

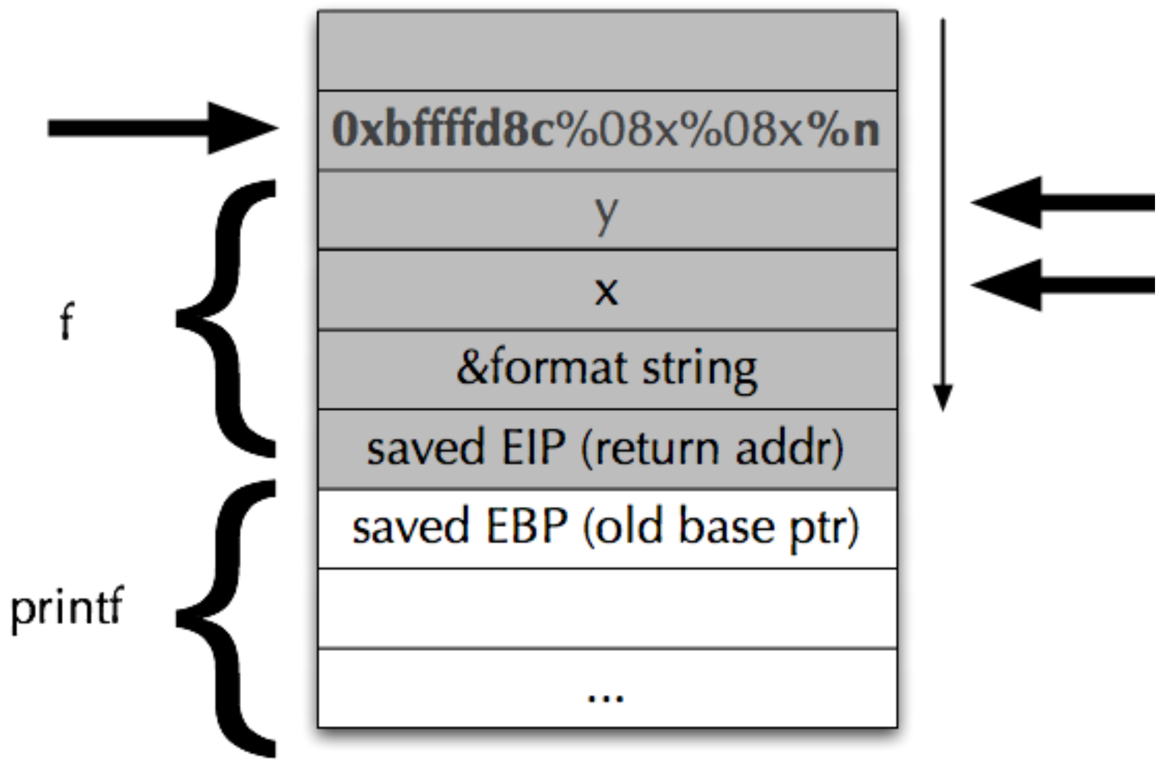
Bad:
```
printf(myString);
```

Format String Errors

Goal:
Craft a special string that can write arbitrary values to arbitrary addresses

Format String Errors

```
f(){
  int x; int y;
  char s[128];
  scanf(s);
  printf(s);
}
```



Heap Overflow

Goal:

Overwrite function pointers on heap to point to injected code

Heap Overflow

- C++ objects are allocated on the heap
- Addresses of these object's functions stored on the heap (vfptr's)
- Overflow heap variable and overwrite these vfptr's
- When function is invoked, our code is executed instead

How do we defend ourselves?

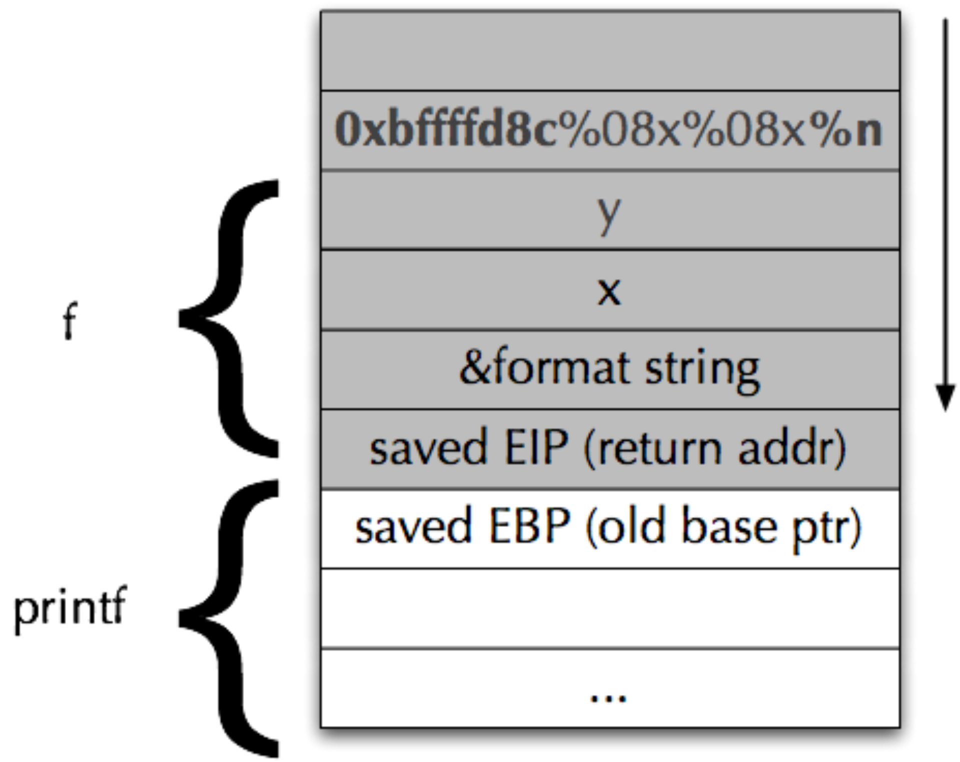
- Canary
- Library Wrapper
- Shadow Stack
- W$\oplus$X Pages

Canary

- Place “Canary” before return address
  - terminator (0x00, 0x0a, 0x0d)
  - random
- Check validity of Canary before returning

Canary (2)

- This is a great solution, right?
- Wrong! What about format string attacks?



Library Wrappers (libsafe)

- Replace know vulnerable function calls with ‘safe’ versions
- ‘Safe’ versions ensure nothing is written past the current stack frame

Library Wrappers (libsafe)

- If we can not get past the stack frame, we can't exploit anything?
- Many problems:
  - User written input loops not protected
  - We can still corrupt local variables
  - We can still do a heap overflow

Shadow Stacks

- Keeps extra copy of return address in separate memory space
- Only allows a return if address matches up

Shadow Stacks (2)

- So, this is the foolproof solution?
- Limitations: Does not protect other data
  - Local variables
  - Heap overflow overwrites function pointers

W$\oplus$X Pages

- *Idea*: if memory is writable, it should not be executable
- Does not allow stack to be executed
  - Try to thwart Stack-smashing

W$\oplus$X Pages

- Game over, we can not execute injected code
- Wait! We can return-to-libc instead

Defense Conclusions

- No defense protects against all memory exploits
- We need a defense-in-breadth approach

Two Countermeasures

- Instruction Set Randomization
- Address Space Randomization

Countering Code-Injection Attacks With Instruction-Set Randomization

Gaurav S. Kc et. Al.
10th ACM International Conference on Computer and Communications Security (CCS)

Intrusion detection: Randomized instruction set emulation to disrupt binary code injection attacks

Elena Gabriela Barrantes et. Al.
10th ACM International Conference on Computer and Communications Security (CCS)

Instruction Set Randomization

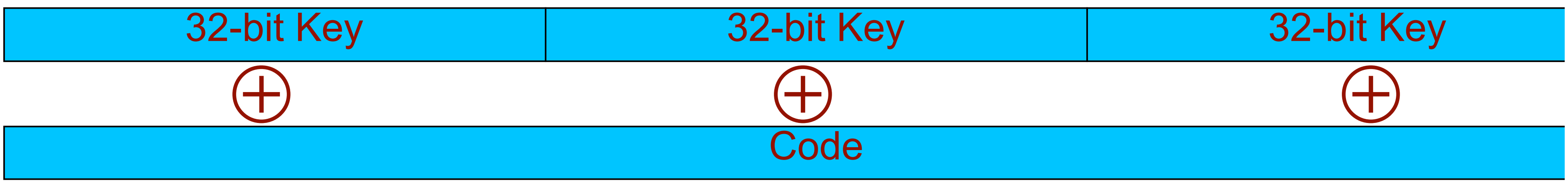
- *Observation*: attackers need to know the instruction set
- *Idea*: Obfuscate the instruction set

How do we obfuscate?

- Encode the machine code of an executable
- Decode instructions before sending to processor

Encoding Process

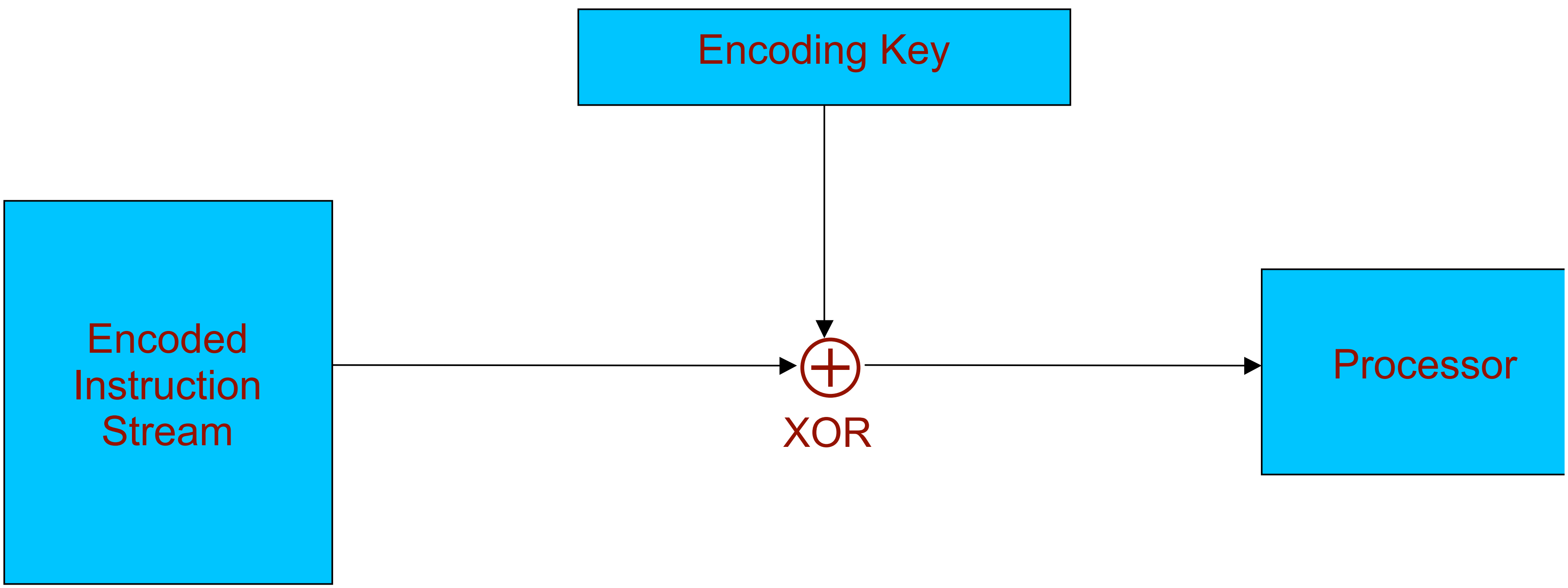
- XOR a key with instructions
    - Worst case for attacker: 2^32 guesses



Barrantes et. al. Proposed a one time pad to the code

Decoding Process

- Decoding is performed when instructions are fetched from memory



Practical Considerations

- Shared libraries
- Kc et al. implemented in hardware (ideally)
- Barrantes et al. implemented in emulator
- Performance may suffer

ISR Thwarts an Attack

0-day exploit

shellcode[] =

```
"\x31\xdb"       // xorl
"\x8d\x43\x17"// leal
"\xcd\x80"       // int
...              //...
```

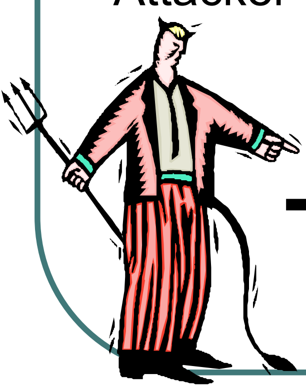
Encoded:

```
"\x31\xdb"       // xorl
"\x8d\x43\x17"// leal
"\xcd\x80"       // int
...              //...
```

Decoded:

```
“\x23\x54”       //invalid
“\xa3\x2f\x9e” //invalid
“\x65\xc1        //invalid
```

Attacker



X86 Apache Web Se
ISR Protected

Crash!

ISR Conclusions

- *The good*: completely eliminates executing injected code, seemingly
- *The bad*: do not always have to inject code

Wheres the FEEB?
On the Effectiveness of Instruction Set Randomization

N. Sovarel, D. Evans, and N. Paul

USENIX Security, 2005

On the Effectiveness of ISR

- *ISR designed to prevent successful* code injection
- But, Sovarel et al. demonstrate attacks that CAN inject code successfully

Assumptions

- Address of vulnerable buffer is known
- Same randomization key used for each forked process
- Encoding vulnerable to known ciphertext-plaintext attack
  - XOR encoding satisfies this assumption
- X86 instruction set is used

Attack Methodology

Goal:
Distinguish between correct and incorrect guesses

Attack Methodology

Encoded
Guess:

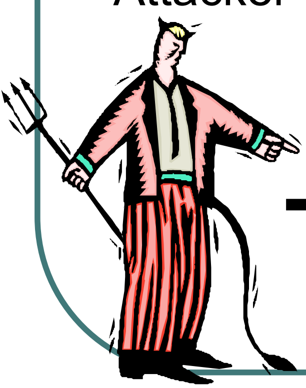
\x01 //ret?
\x02 //ret?
\x03 //ret?
\x04 //ret?
\x05 //ret?

Decoded:

“\x23\x54” //invalid

\xc5 //invalid (crash)
\xef //invalid (crash)
\x7a //invalid (crash)
\xc3 //valid
(observable behavior)

Attacker



X86 Apache Web Se
ISR Protected

ISR Attacks

- Return attack
- Jump attack
- Extended attack

Return Attack

- Inject a 1-byte near return instruction
- Incorrect guess causes a crash
- Correct guess causes observable behaviour
  - For example, some output will be returned

Return Attack (2)

Top of stack
…
Local Buffer
…
Return address
…
Bottom of stack

Normal Stack Layout

Top of stack
…
Local Buffer
Near return (0xc3)
…
Address of buffer
Original return address
Bottom of stack

Stack Layout After Attack

Several Hurdles to Jump

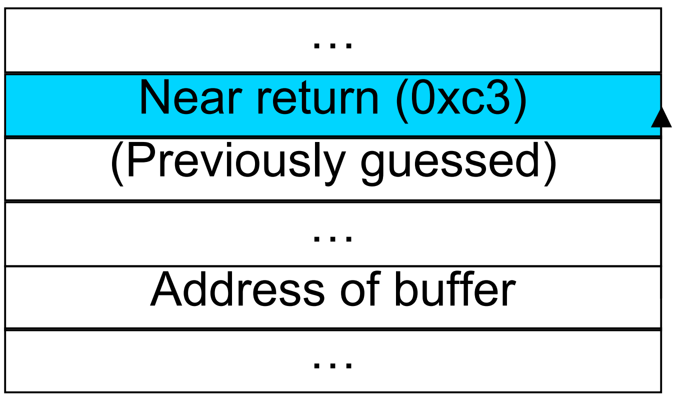
- The stack has been corrupted
- What about false positives?

False Positives

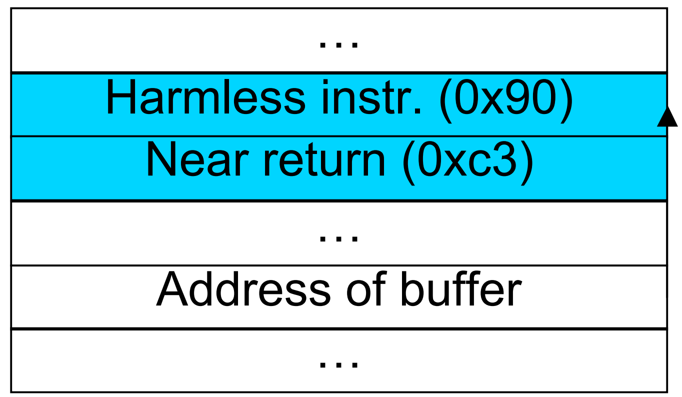
- Apparent correct behavior in several circumstances:
  - It was actually correct (1/256)
  - Another opcode produced the same behavior; 'near return and pop' instruction (1/256)
  - It decoded to a harmless opcode (NOP, etc), and some other instruction produced the same behavior

Reducing False Positives (2)

Use a harmless instruction to eliminate false positives



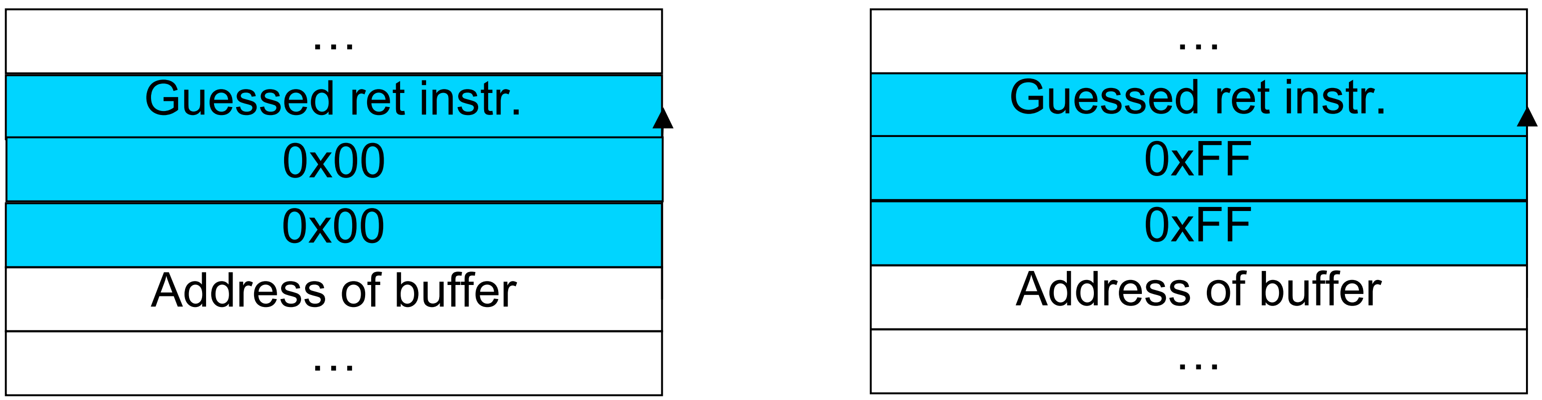
(1) Apparently correct



(2) Double check

Reducing False Positives (3)

- Near return / near return and pop very similar



(1) Apparently Correct

(2) Double Check

Return Attack Conclusions

- *Strength*: only need to guess a 1-byte instruction at a time
- *Weakness*: stack corruption makes it difficult to use reliably

Jump Attack

Jump Attack

- Inject a 2-byte short jump instruction
- Correct guess causes an infinite loop
- Incorrect guess causes crash

Jump Attack (2)

Top of stack
…
Local Buffer
…
Return address
…
Bottom of stack

Normal Stack Layout

Top of stack
…
Local Buffer
Short jump (0xeb)
Offset (0xfe)
…
Address of buffer
…
Bottom of stack

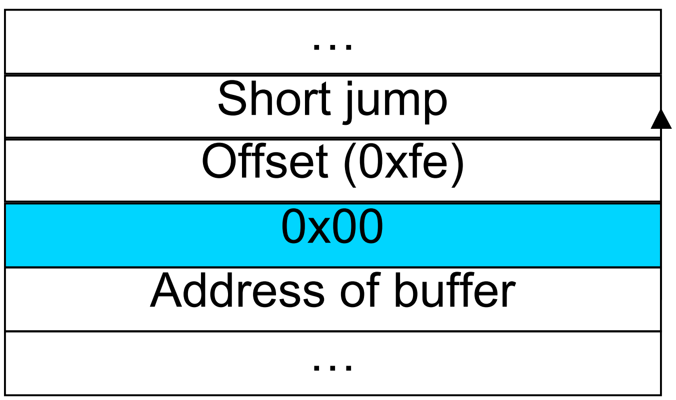
Stack Layout After Attack

False Positives

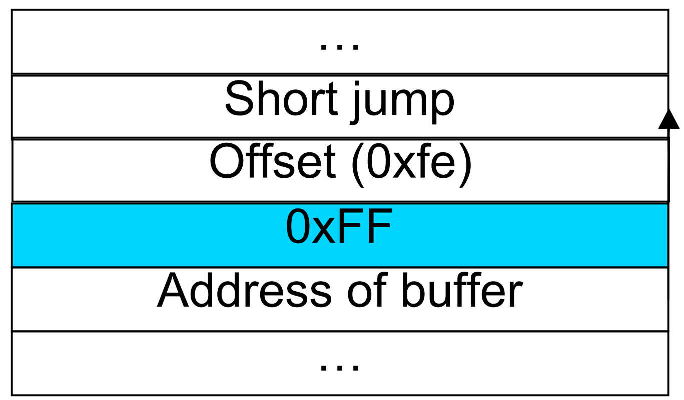
- Again, apparent correct behavior will be exhibited in several circumstances:
  - It was actually correct
  - An incorrectly decoded instruction produced an infinite loop; there are 16 near conditional jumps
  - It decoded to a harmless instruction (NOP, etc), and some other instruction produced an infinite loop

False Positives (2)

Change high bit in the 3rd byte to eliminate false positives



(1) Apparently Correct



(2) Double Check

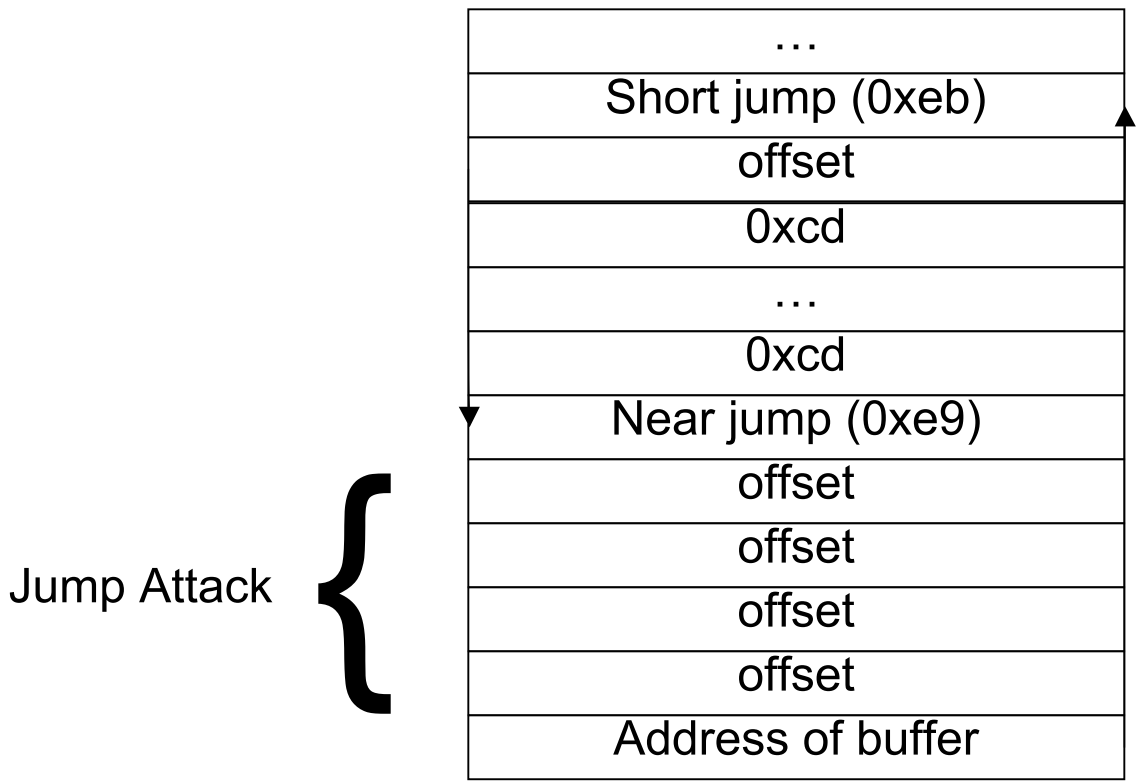
Jump Attack Conclusions

- *Strength*:
  - Use not restricted to special circumstances
- *Weaknesses*:
  - 2-byte instruction must be guessed
  - Infinite loops created

Extended Attack

Extended Attack

- Near jmp jumps to original return address



Stack Layout After Attack

Extended Attack Conclusions

- *Strengths*:
  - Not restricted to special circumstances
  - Only creates a few infinite loops
- *Weaknesses*:
  - Initially 2-byte instructions must be guessed

MicroVM

- Consider an ISR aware worm
- Proposed ‘MicroVM’ is only 100 bytes long
  - Use to execute small chunks of the worm at a time

Results

- Is 6 minutes and 8,636 attempts reasonable?

Key Bytes	Attempts	Attempts per byte	Infinite Loops	Success Rate (%)	Time (s)
2	3983	1991.6	3.86	98	138.3
4	4208	1052.1	8.11	99	207.9
32	7240	226.3	8.28	98	283.6
100	8636	86.4	9.15	100	365.6
512	18904	36.9	8.31	95	627.4
1024	30035	29.3	7.90	100	947.3
4096	102389	25.0	8.36	95	2919.4

Practical Considerations

- The attacks make many assumptions
  - Address of buffer is known
  - Key is not re-randomized
  - Encoding vulnerable to known plaintext-ciphertext attack
- Attacks are x86 instruction set dependent

Wheres the FEEB? Conclusions

- ISR can easily fix the assumptions
  - In fact, Sovarel et. al. had to change the RISE implementation to conform
- Take this paper as a lesson in safe implementation

”if you’re going to implement ISR, make sure ever process gets a fresh key!”

“When I have tried to exploit buffer overflows, a noop sled has always been needed”

ISR Conclusions

- *The good* – Effectively eliminates code injection, if implemented correctly
- *The bad* – Implemented in hardware or an emulator
- *The ugly* – Still, does nothing to protect against return-to-libc

We still need a more general approach!

Address Space Randomization

Address Space Randomization

- *Observation*: Attacker needs to know certain addresses in memory
- *Idea*: Obfuscate memory addresses

PaX ASLR

- PageEXec Address Space Layout Randomization brought to us by the PaX Team
- Popular open-source ASR implementation
  - Hardened Debian
  - Hardened Gentoo
  - Grsecurity kernel enhancements
- Randomizes: stack, heap, libraries

PaX - Randomization

32-bit architecture process address space

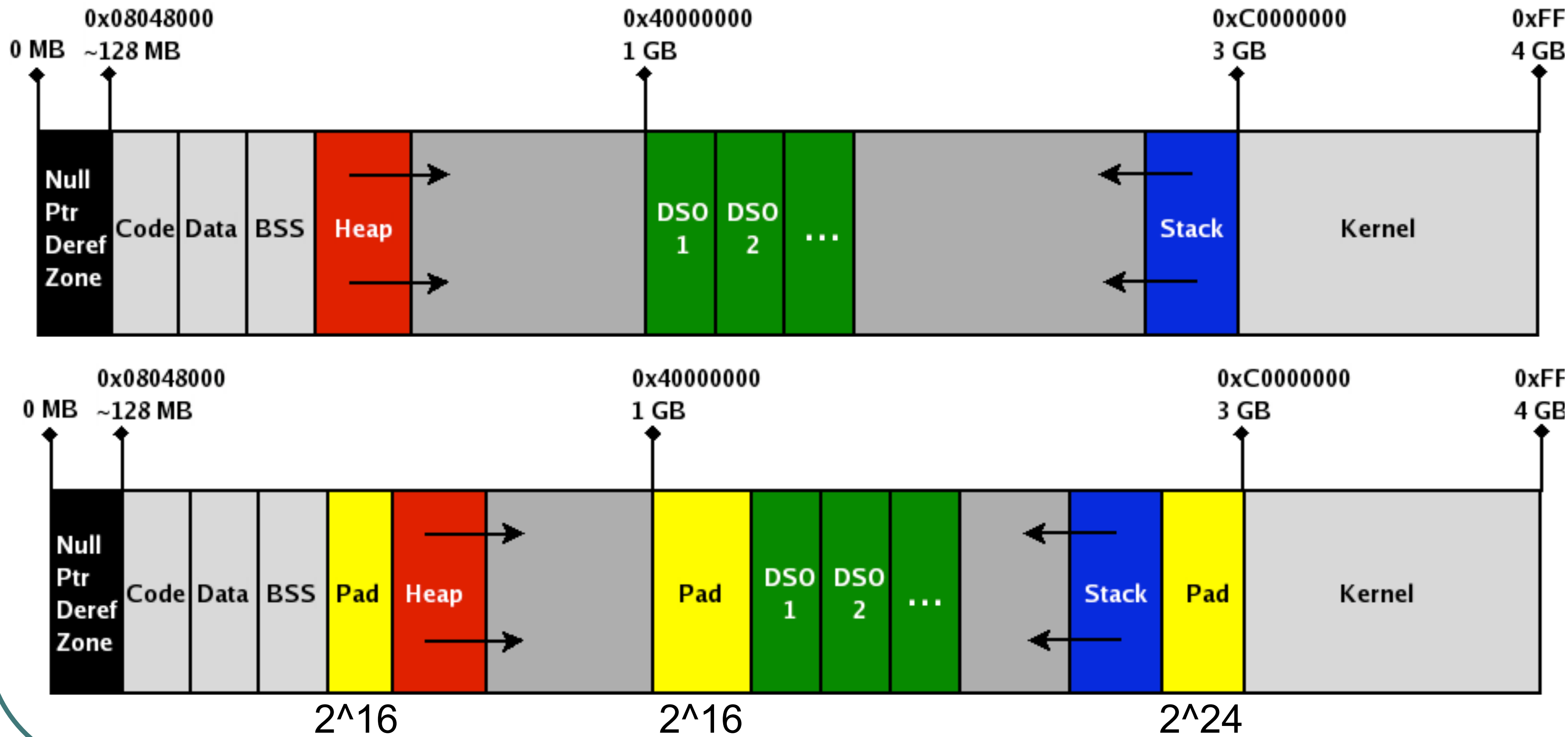


Image Source: http://www.csc.ncsu.edu/faculty/junxu/software/

On the Effectiveness of Address-Space Randomization

- *Goal*: Guess library offset and compute location of `system()`
- *Means*: return-to-libc, lack of entropy in PaX ASLR randomization

The Exploit

- Note:
  - Library offset is limited to 2^{16} possibilities
  - PaX ASLR does not rerandomize on `fork()`
  - Relative addresses inside libraries are not randomized

The Exploit - Setup

- Apache web server on a 32-bit architecture
- PaX ASLR for randomization
- Separated attack machine from victim with a 100 Mbps network

The Exploit - Guessing Addresses

- Send probes using return-to-libc attack
- Unsuccessful guess crashes
- Successful guess produces observable behavior

TIME: `usleep()`

Attack Methodology

Offset?

0x00000001

0x00000002

0x00000003

0x00000004

Result:

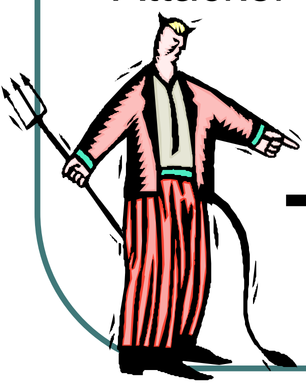
Crash!

Crash!

Crash!

Sleep 16 seconds

Attacker



X86 Apache Web Se

ASR Protected

Probing for the offset

Top of stack
…
…
Arguments
Return address
…
64 byte buffer
…
Bottom of stack

Normal Stack Layout

Top of stack
…
Arg (0x01010101)
Ret (0xDEADBEEF)
Usleep() addr.
AAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAA
…
Bottom of stack

Stack Layout After Attack

Return-to-libc Attack

Top of stack
Buffer addr.
…
…
…
Arguments
Return address
…
64 byte buffer
…
Bottom of stack

Normal Stack Layout

Top of stack
Buffer addr.
Ret (0xDEADBEEF)
System() addr.
Ret() addr.
Ret() addr.
Ret() addr.
AAAAAAAAAAAAAAAAA
'/bin/sh'
…
Bottom of stack

Stack Layout After Attack

ASR Conclusions

- *The good*: attempts to hinder all types of memory exploits (defense-in-breadth)
- *The bad*: low entropy leaves it vulnerable

We can still do better!

A better approach to ASR?

- 64-bit architecture
  - Can increase randomness from 2^16 to 2^40
- Randomization Frequency
- Granularity
  - Permute stack variables
  - Permute code & library functions
  - Permute static data
- Combine with other approaches

Questions?

References

Wheres the FEEB?
On the Effectiveness of
Instruction Set Randomization

N. Sovarel, D. Evans, and N. Paul

USENIX Security, 2005

References

On the Effectiveness of Address Space Randomization

H. Schacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, D. Boneh

ACM CCS 04

References

Countering Code-Injection Attacks With Instruction-Set Randomization

Gaurav S. Kc et. Al.

10th ACM International Conference on Computer and Communications Security (CCS)

Intrusion detection: Randomized instruction set emulation to disrupt binary code injection attacks

Elena Gabriela Barrantes et. Al.

10th ACM International Conference on Computer and Communications Security (CCS)

References

- Thanks to Lucas Ballard for lending some of his slides for this presentation.
- Images on slide 64 are from the Address Space Layout Permutation project by Jun Xu at North Carolina State Univ.