hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
hashOut.length = SSL_SHA1_DIGEST_LEN;
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

err = sslRawVerify(...);
CSE 484: Computer Security and Privacy

Software Security: More!

Spring 2024

David Kohlbrenner
dkohlbre@cs

Thanks to Franzi Roesner, Dan Boneh, Dieter Gollmann, Dan Halperin, David Kohlbrenner, Yoshi Kohno, Ada Lerner, John Manferdelli, John Mitchell, Vitaly Shmatikov, Bennet Yee, and many others for sample slides and materials ...
Logistics

• HW1 due **tonight**

• 584 reading 1 due Friday

• Lab 1 is running
Last time...

- Stack smashing and overwriting return pointers

- “Computing” with printf
Viewing Memory

• %x format symbol tells printf to output data on stack
  ```c
  printf("Here is an int:  %x", i);
  ```

• What if printf does not have an argument?
  ```c
  char buf[16]="Here is an int:  %x";
  printf(buf);
  ```

• Or what about:
  ```c
  char buf[16]="Here is a string:  %s";
  printf(buf);
  ```
Viewing Memory

• %x format symbol tells printf to output data on stack

```c
printf("Here is an int:  %x", i);
```

• What if printf does not have an argument?

```c
char buf[16]="Here is an int:  %x";
printf(buf);
```

– Stack location pointed to by printf’s internal stack pointer will be interpreted as an int.
(What if crypto key, password, ...?)

• Or what about:

```c
char buf[16]="Here is a string:  %s";
printf(buf);
```

– Stack location pointed to by printf’s internal stack pointer will be interpreted as a pointer to a string
Writing Stack with Format Strings

- `%n` format symbol tells `printf` to write the number of characters that have been printed

  ```c
  printf("Overflow this!\%n", &myVar);
  ```
  - Argument of `printf` is interpreted as destination address
  - This writes 14 into `myVar` ("Overflow this!" has 14 characters)

- What if `printf` does not have an argument?

  ```c
  char buf[16]="Overflow this!\%n";
  printf(buf);
  ```
  - Stack location pointed to by `printf`’s internal stack pointer will be interpreted as address into which the number of characters will be written.
Summary of Printf Risks

• **Printf** takes a variable number of arguments
  • E.g., `printf(“Here’s an int: %d”, 10);`

• **Assumptions about input can lead to trouble**
  • E.g., `printf(buf)` when `buf=“Hello world”` versus when `buf=“Hello world %d”`
  • Can be used to advance printf’s internal stack pointer
  • Can read memory
    • E.g., `printf(“%x”)` will print in hex format whatever printf’s internal stack pointer is pointing to at the time
  • Can write memory
    • E.g., `printf(“Hello%n”);` will write “5” to the memory location specified by whatever printf’s internal SP is pointing to at the time
How Can We Attack This?

```c
foo() {
    char buf[2048];
    strncpy(buf, readUntrustedInput(), sizeof(buf));
    printf(buf); //vulnerable
}
```

If format string contains % then printf will expect to find arguments here...

What should the string returned by readUntrustedInput() contain?

Different compilers / compiler options / architectures might vary
Using %n to Overwrite Return Address

In foo()'s stack frame:

Buffer with attacker-supplied input “string”

“... attackString%n”, attack code

&RET

SFP

RET

Key idea: do this 4 times with the right numbers to overwrite the return address byte-by-byte.
(4x %n to write into &RET, &RET+1, &RET+2, &RET+3)

Why is “in” in quotes? C allows you to concisely specify the “width” to print, causing printf to pad by printing additional blank characters without reading anything else off the stack.

Example: printf(“%5d%n”, 10) will print three spaces followed by the integer: “ 10”

That is, the %n will write 5, not 2.

When %n happens, make sure the location under printf’s stack pointer contains address of RET; %n will write the number of characters in printed so far into RET

Number of characters “in” attackString must be equal to ... what?

This portion contains enough % symbols to advance printf’s internal stack pointer
The exploitation twilight zone

• During an exploitation attempt sometimes you have to ‘let it run’
  • Overflow a buffer
  • Change things
  • Let program run for ‘a bit’
  • Everything triggers!

• Printf exploit a perfect example
Recommended Reading

• It will be hard to do Lab 1 without:
  • Reading (see assignments):
    • Smashing the Stack for Fun and Profit
    • Exploiting Format String Vulnerabilities
Buffer Overflow: Causes and Cures

• Classical memory exploit involves code injection
  • Put malicious code at a predictable location in memory, usually masquerading as data
  • Trick vulnerable program into passing control to it

• Possible defenses:
  1. Prevent execution of untrusted code
  2. Stack “canaries”
  3. Encrypt pointers
  4. Address space layout randomization
  5. Code analysis
  6. Better interfaces
  7. ...
Defense: Better string functions!

• `strcpy` is bad
• `strncpy` is... also bad (no null terminator! Returns `dest`!)

4/2/2024  CSE484 - Spring 2023
Defense: Better string functions!

• strcpy is bad
• strncpy is... also bad (no null terminator! Returns dest!)
• BSD to the rescue: strlcpy
  • size_t strlcpy(char *dest, const char *src, size_t n);
    • Always NUL terminates
    • Returns len(src) ...

Ushering out strlcpy()

By Jonathan Corbet  August 25, 2022

With all of the complex problems that must be solved in the kernel, one might think that copying a string would draw little attention. Even with the hazards that C strings present, simply moving some bytes should not be all that hard. But string-copy functions have been a frequent subject of debate over the years, with different variants being in fashion at times. Now it seems that the BSD-derived strlcpy() function may finally be on its way out of the kernel.
ASLR: Address Space Randomization

• Randomly arrange address space of key data areas for a process
  • Base of executable region
  • Position of stack
  • Position of heap
  • Position of libraries

• Introduced by Linux PaX project in 2001
• Adopted by OpenBSD in 2003
• Adopted by Linux in 2005
ASLR: Address Space Randomization

- Deployment (examples)
  - Linux kernel since 2.6.12 (2005+)
  - Android 4.0+
  - iOS 4.3+ ; OS X 10.5+
  - Microsoft since Windows Vista (2007)

- Attacker goal: Guess or figure out target address (or addresses)
- ASLR more effective on 64-bit architectures
Attacking ASLR

- **NOP sleds** and **heap spraying** to increase likelihood for adversary’s code to be reached (e.g., on heap)

- Brute force attacks or memory disclosures to map out memory on the fly
  - Disclosing a single address can reveal the location of all code within a library, depending on the ASLR implementation
Defense: Executable Space Protection

• Mark all writeable memory locations as non-executable
  • Example: Microsoft’s Data Execution Prevention (DEP)
  • This blocks many code injection exploits

• Hardware support
  • AMD “NX” bit (no-execute), Intel “XD” bit (execute disable) (in post-2004 CPUs)
  • Makes memory page non-executable

• Widely deployed
  • Windows XP SP2+ (2004), Linux since 2004 (check distribution), OS X 10.5+ (10.4 for stack but not heap), Android 2.3+
What Does “Executable Space Protection” Not Prevent?

• Can still corrupt stack …
  • … or function pointers
  • … or critical data on the heap

• As long as RET points into existing code, executable space protection will not block control transfer!
  ➔ return-to-libc exploits
return-to-libc

• Overwrite saved ret (IP) with address of any library routine

• Does not look like a huge threat?
  • ...

• Gradescope time
return-to-libc

• Overwrite saved ret (IP) with address of any library routine
  • Arrange stack to look like arguments

• Does not look like a huge threat
  • ...
  • We can call *any* function we want!
  • Say, exec 😊
return-to-libc++

• Insight: Overwritten saved EIP need not point to the *beginning* of a library routine
• **Any** existing instruction in the code image is fine
  • Will execute the sequence starting from this instruction
• What if instruction sequence contains RET?
  • Execution will be transferred... to where?
  • Read the word pointed to by stack pointer (SP)
    • Guess what? Its value is under attacker’s control!
  • Use it as the new value for IP
    • Now control is transferred to an address of attacker’s choice!
  • Increment SP to point to the next word on the stack
Chaining RETs

- Can chain together sequences ending in RET
  - Krahmer, “x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique” (2005)

- What is this good for?

- Answer [Shacham et al.]: everything
  - Turing-complete language
  - Build “gadgets” for load-store, arithmetic, logic, control flow, system calls
  - Attack can perform arbitrary computation using no injected code at all – return-oriented programming

- Truly, a “weird machine”
Return-Oriented Programming
Defense: Run-Time Checking: StackGuard

• Embed “canaries” (stack cookies) in stack frames and verify their integrity prior to function return
  – Any overflow of local variables will damage the canary

[Diagram of stack frames with labels for buf, canary, sfp, ret addr, and frame of the calling function.]

Top of stack

Local variables

Pointer to previous frame

Return execution to this address
**Defense: Run-Time Checking: StackGuard**

- Embed "canaries" (stack cookies) 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

- Canary contains: "\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 at one point in time
Defeating StackGuard

• StackGuard can be defeated
  – A single memory write where the attacker controls both the value and the destination is sufficient

• Suppose program contains `copy(buf,attacker-input)` and `copy(dst,buf)`
  – Example: dst is a local pointer variable
  – Attacker controls both buf and dst

\[\text{Return execution to this address}\]

\[\text{Overwrite destination of strcpy with RET position}\]