CE4062/CZ4062
Computer Security

Lecture 4: Software Vulnerability Defenses

Tianwei Zhang

Outline

- **Safe Programing**
- **Vulnerability Detection**
- **Compiler Support**
- **OS Support**

Outline

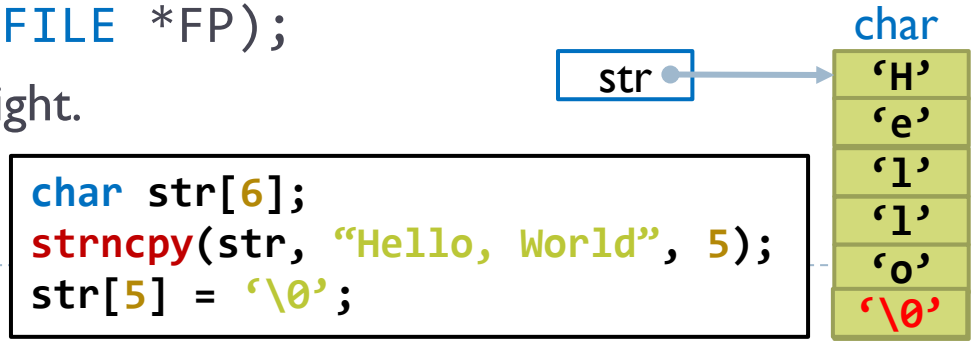
- **Safe Programing**
- Vulnerability Detection
- Compiler Support
- OS Support

Safe functions

Root cause: unsafe C lib functions have no range checking

- `strcpy (char *dest,  char *src)`
- `strcat (char *dest, char *src)`
- `gets (char *s)`

- Use "safe" versions libraries:
  - `strncpy (char *dest, char *src, int n)`
    - Copy *n* characters from string src to dest
    - Does not automatically add the NULL value to dest if *n* is less than the length of string src. So it is safer to always add NULL after `strncpy.`
  - `strncat (char *dest, char *src, int n)`
  - `fgets(char *BUF, int N, FILE *FP);`
  - Still have to get the byte count right.



Assessment of C Library Functions

Extreme risk

- `gets`

High risk

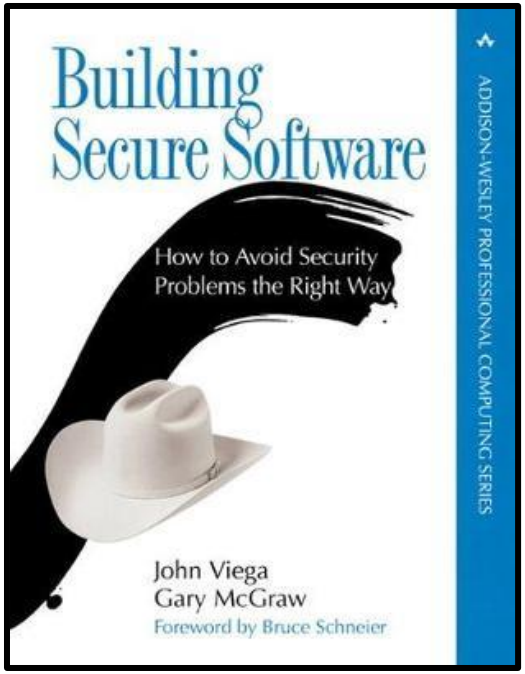
- `strcpy, strcat, sprintf, scanf, sscanf, fscanf, vfscanf, vsscanf, streadd, strecpy, strtrns, realpath, syslog, getenv, getopt, getopt_long, getpass`

Moderate risk

- `getchar, fgetc, getc, read, bcopy`

Low risk

- `fgets, memcpy, snprintf, strccpy, strcadd, strncpy, strncat, vsnprintf`



Safe Libraries

libsafe

- Check some common traditional C functions
  - Examines current stack & frame pointers
  - Denies attempts to write data to stack that overwrite the return address or any of the parameters

glib.h

- Provides Gstring type for dynamically growing null-terminated strings in C

Strsafe.h

- A new set of string-handling functions for C and C++.
- Guarantees null-termination and always takes destination size as argument

SafeStr

- Provides a new, high-level data type for strings, tracks accounting info for strings; Performs many other operations.

Glib

- Resizable & bounded

Apache portable runtime (APR)

- Resizable & bounded

Safe Language

Use Strong type language

- Ada, Perl, Python, Java, C#, and even Visual Basic have automatic bounds checking, and do not have direct memory access
- C-derivatives: Rust (Mozilla 2010):
  - Designed to be a "safe, concurrent, practical language", supporting functional and imperative-procedural paradigms
  - Does not permit null pointers, dangling pointers, or data races
  - Memory and other resources are managed through "Resource Acquisition Is Initialization" (RAII).
- Go: type-safe, garbage-collected but C-looking language
  - Good concurrency model for taking advantage of multicore machines
  - Appropriate for implementing server architectures.

Outline

- Safe Programing
- **Vulnerability Detection**
- Compiler Support
- OS Support

Manual Code Reviews

Peer review

- Very important before shipping the code in IT companies

Code review checklist

- Wrong use of data:

  variable not initialized, dangling pointer, array index out of bounds, …
- Faults in declarations

  undeclared variable, variable declared twice, …
- Faults in computation

  division by zero, mixed-type expressions, wrong operator priorities, …
- Faults in relational expressions

  incorrect Boolean operator, wrong operator priorities, …
- Faults in control flow

  infinite loops, loops that execute n-1 or n+1 times instead of n, ...

Software Tests

Unit tests

- Check that each piece of code behaves as expected in isolation
- Unit tests should cover all code, including error handling

Regression tests

- Check that old bugs haven't been reintroduced

Integration tests

- Check that modules work together as expected

Static Analysis

Analyze the source code before running it (during compilation)

- Explore all possible execution consequences with all possible input
- Approximate all possible states
- Limitations: can introduce false positives.

Static analysis tools

- Coverity: https://scan.coverity.com/
- Fortify: https://www.microfocus.com/en-us/cyberres/application-security
- GrammarTech: https://www.grammatech.com/

Fuzzing

Key Idea:

- Find software bugs in a program by feeding it random, corrupted, or unexpected data
- Random inputs will explore a large part of the state space
- Some unintended states are observable as crashes
- Any crash is a bug, but only some bugs are exploitable

A lot of software testing tools based on fuzzing

- AFL: https://github.com/google/AFL
- FOT: https://sites.google.com/view/fot-the-fuzzer
- Peach: https://wiki.mozilla.org/Security/Fuzzing/Peach
- Springfield: https://blogs.microsoft.com/ai/microsoft-previews-project-springfield-cloud-based-bug-detector/

Mutation-based Fuzzing

Steps:

- Collect a corpus of inputs that explores as many states as possible
- Perturb inputs randomly, possibly guided by heuristics
  - Modify: bit flips, integer increments
  - Substitute: small integers, large integers, negative integers
- Run the program on the inputs and check for crashes

```
numwrites = random.randrange(math.ceil((float(len(buf)) / FuzzFactor))) + 1
for j in range(numwrites):
    rbyte = random.randrange(256)
    rn = random.randrange(len(buf))
    buf[rn] = "%c"%(rbyte)
```

Pros

- Simple to set up;
- Use off-the-shelf software for many programs.

Cons

- Results depend on the quality of the initial corpus;
- coverage may be shallow.

Generation-based Fuzzing

Steps:

- Convert a specification of the input format (RFC, etc.) into a generative procedure
- Generate test cases according to the procedure and introduce random perturbations
- Run the program on the inputs and check for crashes

Pros

- Get higher coverage by leveraging knowledge of the input format;

Cons

- Requires lots of effort to set up;
- Domain-specific;

Coverage-guided Fuzzing

Steps:

- Using traditional fuzzing strategies to create new test cases by mutating the input
- Test the program and measure the code coverage.
- Using the code coverage as a feedback to craft input for uncovered code

Pros

- Good at finding new program states;
- Combine well with other fuzzing techniques;

Cons

- Cannot find all types of bugs

Outline

- Safe Programing
- Vulnerability Detection
- **Compiler Support**
- OS Support

StackGuard

Key insight

- It is difficult for attackers to only modify the return address without overwriting the stack memory in front of the return address.

Steps

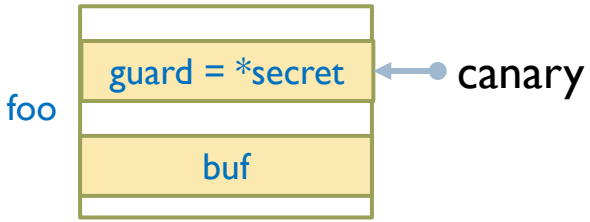
- Embed a canary word next to the return address on the stack whenever a function is called.
  - The value of canary needs to be random and cannot be guessed by the attacker.
- When a stack-buffer overflows into the function return address, the canary is as well overwritten
- Every time the function returns, check whether the canary value has been changed.
- If so, stack-buffer overflows happens, and abort the program.

First introduced as a set of GCC patches in 1998

How does StackGuard Work

```
void foo(char *s) {




  char buf[12];
  strcpy(buf,s);




}
```

```
void foo(char *s) {
  int guard;
  int *secret = malloc(size_of(int));
  *secret = generateRandomNumber();
  guard = *secret;

  char buf[12];
  strcpy(buf,s);

  if (guard == *secret)
    return;
  else
    exit(1);
}
```



Canary Type

Random Canary

- Choose random string at program startup
- Insert canary string into every stack frame.
- Verify canary before returning from function.

  If canary value is changed, then exit program (potential Denial-of-Service attack)
- To corrupt, attacker must learn current random string

Terminator canary.

- Canary = {0, newline, linefeed, EOF}
- String functions will not copy beyond terminator
- Attacker cannot use string functions to corrupt stack.

Limitations of StackGuard

Efficiency

- Program must be recompiled.
- Minimal performance effects: 8% for Apache.

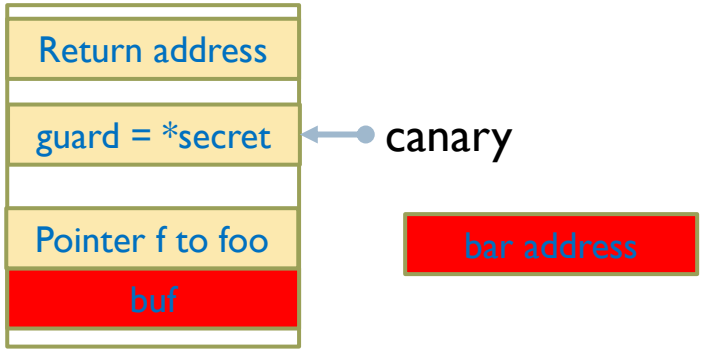
Canaries don't offer full-proof protection

- Some stack smashing attacks can leave canaries untouched.

Hijacking a function pointer

- Even if the attacker cannot overwrite the return address due to the canary, he can overwrite a function pointer.

```
void foo () {...}
void bar () {...}
int main() {
    void (*f) () = &foo;
    char buf [16];
    gets(buf);
    f();
}
```



PointGuard

Protect the pointers from being overwritten

- Encrypt all pointers while in memory
  - Generate a random key when program is executed
  - Each pointer is XORed with this key when stored into memory
- Attacker cannot predict the target program's key
  - Even if the pointer is overwritten, after XORing with the key it will point to a "random" memory address.
  - This can prevent the execution of malicious functions, but can crash the program: denial-of-Service attack
- More noticeable performance overhead

StackShield

Idea

- A GNU C compiler extension that protects the return address.
- Separate control (return address) from data.
  - When a function is called, StackShield copies away the return address to a non-overflowable area.
  - Upon returning from a function, the return address is stored.
  - Therefore, even if the return address on the stack is altered, it has no effect since the original return address will be copied back before the returned address is used to jump back.

Outline

- Safe Programing
- Vulnerability Detection
- Compiler Support
- **OS Support**

Control Flow Integrity (CFI)

Software execution must follow a path of a Control Flow Graph (CFG) determined ahead of time

Direct jump

- The destination address is constant, and easy to check

Indirect jump

- Destination address is determined at runtime, which cannot be predicted ahead of time.
- Prepare a set of allowed destination addresses. At runtime, check whether the target address falls into this list
- Cannot prevent attacks that cause a jump to a valid but wrong address.
- Building accurate CFG statically is hard.

Address Space Layout Randomization

ASLR

- The attacker needs to get the address of their injected code.
- The OS can randomly arrange address space of key data areas for each program
  - *Base of executable region*
  - *Position of stack*
  - *Position of heap*
  - *Position of libraries*
- Make it harder for the attacker to get the address
- Functions remain correct if the stack and base pointers are set up correctly

Deployment

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

ASLR Example

```c
#include <stdio.h>
#include <stdlib.h>
void main() {
    char x[12];
    char *y = malloc(sizeof(char)*12);
    printf("Address of buffer x (on stack): 0x%x\n", x);
    printf("Address of buffer y (on heap) : 0x%x\n", y);
}
```

```
$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
$ a.out
Address of buffer x (on stack): 0xbffff370
Address of buffer y (on heap) : 0x804b008
$ a.out
Address of buffer x (on stack): 0xbffff370
Address of buffer y (on heap) : 0x804b008
```

```
$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
$ a.out
Address of buffer x (on stack): 0xbf9c76f0
Address of buffer y (on heap) : 0x87e6008
$ a.out
Address of buffer x (on stack): 0xbfe69700
Address of buffer y (on heap) : 0xa020008
```

Non-Executable Memory

Mark all writeable memory locations as non-executable

- Attackers inject the malicious code into the memory, and jump to it.
- Configure the writable memory region to be non-executable, and thus preventing the malicious code from being executed.
- Windows: Data Execution Prevention (DEP)
- Linux: ExecShield

```
# sysctl -w kernel.exec-shield=1   // Enable ExecShield
# sysctl -w kernel.exec-shield=0   // Disable ExecShield
```

Hardware support

- AMD64 (**NX-bit**), Intel x86 (**XD-bit**), ARM (**XN-bit**)
- Each Page Table Entry (PTE) has an attribute to control if the page is executable

Limitations: JIT

Two types of executing programs

- Compile a program to a native binary code, and then executes it on a machine (C, C++)
- Use an interpreter to interpret the source code and then execute it (Python)

Just-in-Time (JIT) compilation

- Compile heavily-used ("hot") parts of the program (e.g., methods being executed several times), while interpret the rest parts.
- Exploit runtime profiling to perform more targeted optimizations than compilers targeting native code directly

This requires executable heap

- Conflict with the Non-executable memory protection

Limitations: Code Reuse Attacks

Non-executable Memory protection does not work when the attacker does not inject malicious code into the stack.

- Then how to compromise the program execution?

Code Reuse attack

- Reuse the code in the program itself without injecting the code

Return-to-lib:

- Replace the return address with the address of dangerous library function
- When function returns, the dangerous library function will execute
- Can chain multiple library function.

Shadow Stack

Keep a copy of the stack in memory

- On call: push ret-address to shadow stack on call.
- On ret: check that top of shadow stack is equal to ret-address on stack.
- If there is difference, then attack happens and the program will be terminated.

Shadow stack with Intel CET

- New register SSP: shadow stack pointer
- Shadow stack pages marked by a new "shadow stack" attribute: only "call" and "ret" can read/write these pages

ARM Memory Tagging Extension (MTE)

Tagging the pointer and memory

- Every 64-bit memory pointer *P* has a 4-bit tag
- Every 16-byte user memory region *R* has a 4-bit tag
- When *P* reads *R*, the processor checks if their tags match
  - Yes: allow access
  - No: hardware exception

Tag match

Tag mismatch.
Hardware exception

```
char *p = malloc(48);

p[33] = 'a';

p[52] = 'a';
```