

Session 05

Defense and Mitigation

Security of Information Systems (SIS)

Computer Science and Engineering Department

November 1, 2023

- ▶ attack: exploit vulnerabilities
- ▶ defense: prevent attacks, make attacks difficult, confine attacks
- ▶ attacker needs to find one security hole
- ▶ defender has to protect all security holes
- ▶ attacker invests time
- ▶ defense mechanisms incur overhead

1 / 49

2 / 49

Papers

- ▶ Mitigating Program Security Vulnerabilities: Approaches and Challenges
- ▶ Securing Web Application Code by Static Analysis and Runtime Protection

3 / 49

Attacker Goals

- ▶ control
- ▶ cripple
- ▶ steal

5 / 49

Exploit

- ▶ determine entries/input
- ▶ graph/automaton describes system/application behavior
- ▶ subvert graph
 - ▶ add new nodes (inject)
 - ▶ add new edges
 - ▶ use existing paths in a different way

6 / 49

Attack Vector

- ▶ chain together multiple exploits
- ▶ gain access, gain privileged access, cripple, steal
- ▶ use vulnerabilities in software, system, web

7 / 49

Time

- ▶ always on the attacker side
- ▶ prevent attacks is better than handling attacks

9 / 49

System Components

- ▶ protect everything
- ▶ attacker need only find **one** flaw
- ▶ defense in depth

10 / 49

Prevention

- ▶ preventive/proactive is better than reactive
- ▶ harden system components
- ▶ monitor everything

11 / 49

Security vs Speed

- ▶ any defense mechanism incurs overhead
- ▶ use both offline (check at development time) and online mechanisms (check/harden during run-time)

12 / 49

Handling Complexity

- ▶ automate processes
- ▶ verification and validation
- ▶ check before deployment
- ▶ prioritize critical parts

13 / 49

Monitoring

- ▶ paranoia is a virtue
- ▶ frequent updates
- ▶ be on the lookout for CVEs

14 / 49

Defensive Steps

- ▶ prevent existence and prevent exploitation
- ▶ during development
- ▶ before deployment
- ▶ during deployment: prevent, react, confine

16 / 49

Prevent Existence

- ▶ prevent existence of bugs and vulnerabilities
- ▶ during development and before deployment
- ▶ Secure Software Development
- ▶ secure coding, defensive programming
- ▶ code auditing, code linting
- ▶ fuzzing, symbolic execution

17 / 49

Prevent Exploitation

- ▶ during deployment
- ▶ if vulnerabilities exist, you cannot exploit them
- ▶ either prevent or make it harder for the attacker
- ▶ harden the application, the system

18 / 49

Making Exploitation it Harder

- ▶ randomize
- ▶ obfuscate
- ▶ break application into multiple apps
- ▶ reduce number of inputs (attack surface)

19 / 49

Preventing Exploitation

- ▶ make memory areas inaccessible
- ▶ isolate components
- ▶ harden executable with checker and sanitizers during runtime
- ▶ disadvantage: incurs overhead

20 / 49

Confine

- ▶ more in session 7: Application Confinement
- ▶ **when** the attack happens, reduce damage
- ▶ sandboxing, permissions
- ▶ treat application as potential malware

21 / 49

React

- ▶ monitor applications, system
- ▶ when attack happens, document, make app/system inaccessible
- ▶ patch as soon as possible
- ▶ investigate, prevent future similar attacks

22 / 49

Mindset

- ▶ application is target of attacker
- ▶ input minimization, input validation
- ▶ you deploy an app that may have flaws or may be malware
- ▶ memory disclosure attacks, application control

24 / 49

Goal

- ▶ prevent control flow hijacking
- ▶ prevent memory/information disclosure
- ▶ be on the look for policy flaws that may allow the app to leak information

25 / 49

CFI

- ▶ *Control Flow Integrity*
- ▶ make sure control flow graph is unchanged during run
- ▶ high overhead
- ▶ fine-grained vs coarse grained CFI

26 / 49

Code Pointers

- ▶ critical memory data
- ▶ target for attacker for control flow hijacking
- ▶ function return addresses, function pointers
- ▶ *Code Pointer Integrity* (faster approach to CFI), next lecture

27 / 49

Prevent Vulnerabilities vs Prevent Exploiting vs Make Unlikely vs Confine

- ▶ prevent vulnerabilities: secure coding, verification, fuzzing, symbolic execution, type safety, safe programming languages (later sessions)
- ▶ prevent exploiting: ASan, StackGuard (canaries), SafeStack, CFI, input validation, DEP
- ▶ make unlikely: ASLR, multiple heaps
- ▶ confine: sandboxing, privacy settings, access control settings, SFI (*Software Fault Isolation*) (later sessions)

28 / 49

Stack Guard / Address Sanitizer

- ▶ stack canary, stack protector
- ▶ added at compile time
- ▶ value (canary) placed between buffer and return address
- ▶ overwriting canary is detected and ends the program
- ▶ may leak canary and overwrite it with itself
- ▶ may overwrite other data (without overwriting canary)
- ▶ may overwrite stack guard exit handler
- ▶ Google Address Sanitizer adds multiple checks, albeit at increased overhead

29 / 49

Input Validation

- ▶ assume input is "evil"
- ▶ prevent injection: command injection, SQL injection, shellcode injection
- ▶ prevent attacks such as billion laughs attacks
- ▶ prevent certain patterns, parse input

30 / 49

CFI

- ▶ monitor control graph
- ▶ monitor calls, jumps, branches
- ▶ aim to do it without incurring significant overhead
- ▶ may happen offline

31 / 49

SafeStack

- ▶ store code pointers in a separate stack
- ▶ buffer overflows will not overflow code pointers
- ▶ provide specific methods to access safe stack data

32 / 49

DEP

- ▶ Data Execution Prevention
- ▶ mark writable memory area as non-executable
- ▶ you cannot write and execute, i.e. inject code
- ▶ data, heap, stack are marked with DEP
- ▶ may be bypassed by using a `mprotect()`-like call to update memory area permissions

33 / 49

ASLR

- ▶ Address Space Layout Randomization
- ▶ new memory sections (especially libraries) are loaded at random addresses
- ▶ makes it difficult to find addresses
- ▶ not that effective on i386; useful on x86_64
- ▶ may be bypassed by information leaking

34 / 49

General

- ▶ secure configuration
- ▶ input sanitization
- ▶ trusted connection
- ▶ no vulnerable dependencies

36 / 49

Verification

- ▶ client side
- ▶ server side

37 / 49

Connection

- ▶ HTTPS, SSL/TLS
- ▶ certificate
- ▶ downgrade attacks

38 / 49

Secure HTTP Headers

- ▶ HTTP Strict Transport Security (HSTS)
- ▶ X-Frame-Options
- ▶ X-XSS-Protection
- ▶ X-Content-Type-Options
- ▶ Content-Security-Policy
- ▶ Referrer-Policy
- ▶ Expect-CT

39 / 49

Database protection

- ▶ **sanitize** queries
- ▶ encrypt data at rest
- ▶ encrypt data in transit
- ▶ **sanitize** queries

40 / 49

General System Defense

- ▶ Intrusion Detection System
- ▶ Intrusion Prevention System

42 / 49

Signing

- ▶ secure boot
- ▶ application signing

43 / 49

Sandboxing

- ▶ Mandatory Access Control
 - ▶ SELinux
 - ▶ SMACK
 - ▶ AppArmor
 - ▶ TOMOYO
- ▶ seccomp

44 / 49

Kernel Config

- ▶ CONFIG.HARDENED.USERCOPY
- ▶ CONFIG.FORTIFY_SOURCE
- ▶ CONFIG.RANDOMIZE_BASE (KASLR)
- ▶ CONFIG.KASAN
- ▶ CONFIG_UBSAN
- ▶ In development
 - ▶ KTSAN
 - ▶ KMSAN
- ▶ grsecurity

45 / 49

Defensive Mechanisms

- ▶ prevent existence, prevent exploitation
- ▶ development, before deployment, during deployment
- ▶ input is the root of all evil
- ▶ look out for control flow hijacks, information leaks, malformed input

47 / 49

Keywords

- ▶ vulnerability
- ▶ exploit
- ▶ attack vector
- ▶ prevention
- ▶ isolation
- ▶ CFI
- ▶ code pointer
- ▶ Stack Guard

- ▶ DEP
- ▶ ASLR
- ▶ Address Sanitizer
- ▶ downgrade attacks
- ▶ secure HTTP headers
- ▶ sandboxing
- ▶ Mandatory Access Control

Resources

- ▶ Let's Encrypt
- ▶ Defeating SSL Using Sslstrip
- ▶ OWASP Secure Headers Project