Operating Systems and Program Security

Kc Udonsi

Application Security & Threat Modelling

Common Threats Against Software

- Presence of security bugs "Vulnerabilities"
- Unauthorized modification e.g Backdoors
- Supply chain bugs - Vulnerabilities in dependencies and/or tooling, partners

Why do Vulnerabilities exist ?

- Fundamental oversights in software design. Designed to do the wrong thing a.k.a Design Flaws
- Implementation flaws/bugs relevant to security a.k.a Technical Flaws
- Faulty inter-operation with executing environment a.k.a Operational Flaws
- **Arbitrarily trusting input data, misplaced trust**

Threat Modelling

- Description of system
- Potential threats to the system (threats against CIA)
- Actions that can be taken to mitigate each threat
- Validation of model
- Threat Modelling Manifesto: https://www.threatmodelingmanifesto.org/
- Think about "abuse cases" and what can be done to mitigate those

Secure Programming

- Familiarity with relevant vulnerability classes
- Modularity - separate modules for separate functionalities
- Sanitize, validate, restrict input data even between modules or components (mutual suspicion)
- Be "fault tolerant" by having a consistent policy to handle failure
- Use reputable, security conscious and well maintained libraries
- Adopt good programming practices, be security aware

Software Security Assessment

- Manual, guided or automated audit and security testing
- Security test cases may validate threat mitigation strategies
- Internal or external auditors methodologically review code for design, implementation or operational flaws
  - Vulnerability Rewards Program, Bug Bounties etc
- Fuzz testing can be combined with manual audits to discover vulnerable code paths
- Can be carried out at various stages of the SDLC

Secure Software Development Life Cycle

- Description of subject
- Potential threats to the system
- Actions that can be taken to mitigate each threat
- Validation of model
- Continuous security testing throughout the SDLC "DevSec Ops"
- Think about "abuse cases" and what can be done to mitigate those

Formal Methods of Verification

Mathematical description of the problem

Refinement steps

Proof of correctness

Executable code
or hardware design

Formal Methods of Verification

➡ Examples:

Hardware design (VHDL, Verilog)

✓ Used by semi-conductor companies such as Intel

Critical embedded software (B/Z, Lustre/Esterel)

✓ Urban Transportation
(METEOR Metro Line 14 in Paris by Alstom)

✓ Rail transportation (Eurostar)

✓ Aeronautic (Airbus, Eurocopter, Dassault)

✓ Nuclear plants (Schneider Electric)

Pros and cons of using formal methods

- ✓ Nothing better than a mathematical proof
- ➡ A code "proven safe" is safe

- ◉ Development is time and effort (and so money) consuming
- ➡ Should be motivated by the risk analysis

- ◉ Do not prevent from specification bugs
- ➡ Example of network protocols

Operating System Security

Exploit mitigation, Endpoint Detection and Response (EDRs), Security Policies

Exploit Mitigation

Exploit Mitigation Contd.

- Fortify Source Functions
- Stack Canaries
- Data Execution Prevention / Non-Executable Stack
- Address Space Layout Randomization (ASLR)

Exploit Mitigation Contd.

- Position Independent Executables
- Control Flow Guard
- Application sandboxing
- Non-exhaustive. Often implemented at OS or Compiler

Fortify Source Functions

➡ GCC macro FORTIFY_SOURCE provides buffer overflow checks for unsafe C libraries

`memcpy, mempcpy, memmove, memset, strcpy, stpcpy, strncpy, strcat, strncat, sprintf, vsprintf, snprintf, vsnprintf, gets`

Checks are performed

- some at compile time (compiler warnings)
- other at run time (code dynamically added to binary)

Canaries

- The compiler modifies every function's prologue and epilogue regions to place and check a value (a.k.a a canary) on the stack
- When a buffer overflows, the canary is overwritten. The programs detects it before the function returns and an exception is raised
- Different types:
  - random canaries
  - xor canaries
- Disabling Canary protection on Linux
  `$ gcc ... -fno-stack-protector`
- Bypassing canary protection : *Structured Exception Handling (SEH)* exploit overwrite the existing exception handler structure in the stack to point to your own code

DEP/NX - Non Executable Stack

- The program marks important structures in memory as non-executable
- The program generates an hardware-level exception if you try to execute those memory regions

- This makes normal stack buffer overflows where you set `eip` to `esp+offset` and immediately run your shellcode impossible

- Disabling NX protection on Linux
  `$ gcc ...-z execstack`

- Bypassing NX protection : *Return-to-lib-c* exploit
  return to a subroutine of the lib C that is already present in the process' executable memory

ASLR - Address Space Layout Randomization

- The OS randomize the location (random offset) where the standard libraries and other elements are stored in memory
- Harder for the attacker to guess the address of a lib-c subroutine
- Disabling ASLR protection on Linux
  `$ sysctl kernel.randomize_va_space=0`
- Bypassing ASLR protection : Brute-force attack to guess the ASLR offset
- Bypassing ASLR protection : *Return-Oriented-Programming (ROP)* exploit use instruction pieces of the existing program (called "gadgets") and chain them together to weave the exploit

PIC/PIE - Position Independent Code/Executables

- **Without PIC/PIE**
  code is compiled with absolute addresses and must be loaded at a specific location to function correctly

- **With PIC/PIE**
  code is compiled with relative addressing that are resolved dynamically when executed by calling a function to obtain the return value on stack

Confined execution environment - Sandbox

A sandbox is tightly-controlled set of resources for untrusted programs to run in

➡ Sandboxing servers - virtual machines

➡ Sandboxing programs

- Chroot, Seccomp, AppArmor in Linux
- Sandbox in MacOS
- Application Guard Windows
- Windows Sandbox

➡ Sandboxing applets - Java and Flash in web browsers

Security Policies

Baselining System Security

- OSes strive for secure out-of-the-box
- Granular controls may be required to customize security posture
- Often pushed down as configurations or profiles in enterprise environment
- May include firewall settings, password strength requirements, application installations, removal drive controls, suspicious site access, file download policies etc.

Vulnerability Management

To Patch or Not to Patch …

- Patches often need to be validated
- Risk-based discovery, prioritization and remediation
- Kernel Data Protection (Windows)
- System Coprocessor / Kernel Integrity Protection (MacOS)
- Pointer Authentication Codes (MacOS)
- Code integrity and signing
- Non-exhaustive. Often implemented at OS or hypervisor level (Virtualization Based Security)

Securing the Kernel

Kernel Patch and Exploit Mitigations

- Kernel Self-Protection (Linux)
- Kernel Patch Guard / Patch Protection (KPP) (Windows)
- Kernel Data Protection (Windows)
- System Coprocessor / Kernel Integrity Protection (MacOS)
- Pointer Authentication Codes (MacOS)
- Code integrity and signing
- Non-exhaustive. Often implemented at OS or hypervisor level (Virtualization Based Security)

Endpoint Detection and Response

Endpoint Protection

- Historic anti-virus - signature based detection
- Heuristics and behavioural based detection
- Implemented as an extension to the kernel often with user-space components
- Passive or Active mode, event logging and streaming
- Often featuring a cloud component for incident investigation and security overview
- Still software hence can be contain vulnerabilities

Endpoint Protection

- Mitre Attack Matrix

-