Module Integrity & Temporary Files

Petr Ročkai

November 11, 2019

Overview

- Part 1: Dynamic Linking
- Part 2: Signatures and Trust
- Part 3: Temporary Files
- Part 4: DRM and Code Obfuscation
- Part 5: Homomorphic Cryptosystems

Part 1: Dynamic Linking

Static Linking

- library code is built into the executable
- distributed as `.a` (UNIX) or `.lib` (Windows)
- library is not needed to run the program
- → easy distribution – no external dependencies

Resource Use

- disk space is taken up by many copies of the same code
- so is RAM when programs are loaded (executed)

Static Linking: Vulnerability Management

- each application ships with its own copy of the code
- what if a problem is found in the library?
- each application needs to be updated separately

Detour: How a Linker Works

- programs need addresses of things
  - global variables
  - procedures
- the compiler often does not know the address
- object files (`.o`) contain relocations
- the linker replaces symbols (names) with addresses

Detour: Copy on Write

- multiple running programs share text
- this is because `fork()` does not copy everything
- saves a lot of RAM when many copies of a program run
- implemented using a memory management unit
- works on a page-by-page (4K on x86) basis

Dynamic Linking

- allows a single library to be shared by many programs
- stored in `.so` (UNIX) or `.dll` files (Windows)
- UNIX: `ld.so` implements runtime linking
- part of the linking process done at execution time

Dynamic Linker

- loads all the pieces into memory
- performs relocation in memory
- hands off execution to the program

- this is actually naive and inefficient
- in practice
  - position-independent code
  - lazy binding

Position-Dependent Code

- normal code must be loaded at a fixed address
  - e.g. absolute jump and call instructions
  - direct references to global data
- runtime linker can rewrite those addresses
  - takes too much time
  - we lose sharing

Position-Independent Code

- we want to confine relocations to a small area
  - this is the global offset table (GOT)
  - holds both data and code relocations
- compilers can emit position-independent code
  - use relative addresses when possible
  - use address tables for indirection (GOT, PLT)

Lazy Binding

- do not relocate at load time
- replace inter-library calls with stubs
- the stub asks the linker to relocate
- the linker rewrites the stub with a jump
- unused parts of the code are never relocated

Library Preloading

- the runtime linker can load additional libraries
  - via `LD_PRELOAD` on UNIX
  - `AppInit_DLLs` on Windows
  - `DYLD_INSERT_LIBRARIES` on OS X
- those extra libraries can override functionality
  - useful for hooking into library calls
  - but also compromises the integrity of the application

Plugins

- often implemented using shared libraries
- not linked into the application
- explicitly loaded at runtime
  - using `dlopen` (UNIX) or `LoadLibrary` (Windows)
  - based on the filename
- used via function pointers obtained by name
  - `dlsym` or `GetProcAddress`

Search Path Attacks

- the system needs to find shared libraries to load
- it is usually possible to extend or override this path
  - `LD_LIBRARY_PATH` on UNIX, `PATH` on Windows
  - current directory is also searched on Windows
- only a problem in special circumstances
  - the library is missing in system locations
  - loading based on the `SearchPath` API on Windows

Library Injection

- arrange for your library to be loaded
  - either via preloading
  - or use the same name as a system library
  - and place it where it's found
- hard to do unless the library is missing on the system
- may be easier with plugins

Interposing Calls

- assume your library has been loaded
- the code in the library runs with privileges of the process
- your implementation of the API can do anything
  - log and exfiltrate arguments and return values
  - modify either of those things
  - completely hijack the application
- you can also `dlopen` the correct library
  - and forward calls to the original

Implications

- always make sure you are loading the correct library
- libraries have to be trusted by the application
- malicious library can do anything the process can do
  - e.g. by using global constructors or `DllMain`
  - those get to run before the main app even starts
- it can also turn the app into a trojan and steal secrets

Use Secure Paths

- the default paths are quite secure
- do not try to outsmart the system
  - e.g. by looking up the library yourself
  - especially bad is using `SearchPath` on Windows
  - do not use `LoadLibrary` to check Windows version
- you can explicitly remove the working directory
  - only an issue on Windows use `SetDllDirectory("")`

Side-by-Side with Checksums (Windows)

- the application ships its own copies of DLLs
- designed to avoid “DLL hell”
- lists DLL checksums → avoids injection
- problem: partially defeats code sharing
- problem: vulnerability management again

Part 2: Signatures and Trust

Signatures: Why?

- executable code is very powerful
- often downloaded from the internet
  - a man in the middle is a possibility
  - they could tamper with the application code
  - instant arbitrary code execution / compromise
- it is very important to establish authenticity

Signatures: Hash Functions

- standard cryptographic hash functions (SHA-1 &c.)
- easy to compute for the package you have
- possibly hard to obtain the expected value
  - maybe fetch using HTTPS
  - but web servers are easy to compromise
  - better if you can get it from multiple sources
- usually needs manual verification
  - users are often lazy and generally unreliable
  - almost as bad as no signature at all

Signatures: Keyed Hashes

- Message Authentication Code (HMAC &c.)
- needs a shared secret
- not suitable for standard distribution models
- could be used in per-customer distribution
- also possibly for subsequent updates

Signatures: Asymmetric Crypto

- this is the standard approach
- problem: PKI / trust management
- reduces one problem to another problem
  - software distribution to key distribution
  - but keys are smaller
  - and once obtained, can be used for many packages
- initial keys can be distributed as hardcopies
  - e.g. on read-only installation media
  - or pre-installed on the computer with the OS

Code Signing: Commercial Examples

- Secure Boot
- Java certificates (includes Android)
- Microsoft Authenticode
- Adobe Air certificates
- Microsoft Office and VBA certs
- Apple Developer Program

Example: MS Authenticode

- based on RSA 2048 and SHA-1
- covers Active-X, plugins, executables
- software vendors need to obtain an X.509 certificate
  - also known as Code Signing Digital ID
  - many different CAs issue those
- the signature is embedded in the application
- when downloaded, the system checks the signature
  - any mismatches are reported but may be overridden
  - kernel code (drivers) are refused

Microsoft WHQL

- Windows Hardware Quality Labs
- stricter requirements than generic Authenticode
- testing logs must be submitted to MS
- however: no code review is done by MS
  - WHQL does not imply the drivers are secure
  - it does imply a certain level of quality
- allows distribution through Windows Update

Code Signing: Open Source

- OpenBSD binary distribution & packages
- FreeBSD and NetBSD likewise
- binary Linux distributions
  - Fedora, Debian, Ubuntu, RHEL, CentOS
  - almost every package manager
- source code is also often signed

Trust

- signed ≠ secure ≠ trustworthy
- you need to trust the vendor
  - possibly backed by a legal contract
  - but usually not for off-the-shelf software
- even honest vendors make mistakes
  - vulnerabilities are widespread
- reviewing source code is the only reliable option

Open Source: Collaborative Trust

- many people look at different bits
- if you find something bad, you speak up
- assume it is OK if everyone is silent
- seems to be working well in practice

Open Source: Which Source?

- how to ensure everyone is looking at the same source?
    - source in `git` or similar
    - signed source distribution tarballs
- rate of change: can the readers keep up?

Reproducible Builds

- how to check the binary came from given source?
- rebuilding may change the checksum of the result
- essential for collaborative trust for binary distributions
- <https://reproducible-builds.org>
- alternative: build everything yourself

Security

- assume we trust the vendor
- when are signatures verified?
  - do we need to decompress the package first?
  - maybe even unpack the content
- trust OK only after the signature is verified
  - the header may be malicious if signature is bad

Part 3: Temporary Files

Why Temporary Files?

- data too large to fit in memory
- transferring data to other programs
- named pipes and UNIX domain sockets
- usually not persistent

Creation in C / C++: ISO C

- `FILE *tmpfile()`
  - created in the default system location
  - deleted on close / program exit
  - unique file name (or no file name at all)
  - opened for reading and writing
- `tmpnam()` and `tempnam()`
  - do not use those functions
  - only for compatibility with very old programs

Creation in C / C++: Windows

- `tmpnam_s()` from secure C library
  - not actually secure
  - never use this function with `fopen`
- `tmpfile_s()`
  - like `tmpfile` but different calling convention
  - neither is very useful on Windows (needs admin)

Creation in C / C++: Windows

- use `CREATE_NEW` in `CreateFile()`
- also specify `FILE_FLAG_DELETE_ON_CLOSE`
- possibly also `FILE_ATTRIBUTE_TEMPORARY`
- you can get the filename by using `tmpnam_s`
- try with a new name if `CreateFile` fails

Creation in C / C++: POSIX

- always use `mkdtemp` and `mkstemp`
- both are secure against race attacks
- `mkostemp` on newer systems
    - allows `O_SYNC` and `O_CLOEXEC` to be specified
- `unlink()` the file to get erase-on-exit

Creation in Java

- `File tmp = File.createTempFile`
- do not leave garbage around: `tmp.deleteOnExit()`
- about as secure as `mkstemp()` in C
- needs at least Java 7

Creation in Python

- `import tempfile`
  - then `help(tempfile)` or check online
- how to tell if the API is good?
  - read the documentation
  - does it mention security? race conditions?
  - is it deprecated? is there a warning?

Temporary File Checklist (1)

- do not use them if not necessary
- never store secrets in temporary files
- do not use standard C functions
  - `tmpnam`, `mktemp`, `tempname` are bad
  - `tmpfile` is sometimes OK on UNIX

Temporary File Checklist (2)

- use platform APIs to prevent races
  - `mkstemp`, `mkdtemp`
  - `open` with `O_CREAT` and `O_EXCL`
  - `CreateFile` with appropriate flags
- ensure proper permissions
  - set a restrictive ACL when calling `CreateFile`
  - already taken care of with `mkstemp`

Part 4: DRM and Code Obfuscation

What is DRM?

- Digital Rights Management
- essentially just copy protection
- as old as commercial software
- usually not very successful

Naive DRM

- embed a secret key in the official viewer
- encrypt all content with the secret key
- distribute the encrypted content
- only the official viewer can play it
- but the key is easy to recover

DRM is Hard

- the attacker has complete control over execution
- can use debuggers, analysers, fuzzers, etc.
- embedded keys are easy to spot (high entropy)
- obfuscation can help, but only a little
- once the key is compromised, so is all the content

White-Box Cryptography

- all of the black-box assumptions
  - mainly chosen plaintext attacks
- the attacker can also look at execution
  - even perturb data while the algorithm runs
  - can see the entire memory
  - including any key material
- hard but (maybe) not impossible

- 2002: White-Box Cryptography, AES Implementation
    - initial proposal by Chow et al.
    - based on encrypted networks, broken in 2004
- 2006: White Box Cryptography: A New Attempt
    - Bringer et al., added perturbations
    - broken in 2010
- 2009: A Secure Implementation of White-Box AES
    - different approach by Xiao et al., broken in 2012
- 2011: Protecting White-Box AES with Dual Ciphers
    - broken in 2013 by CRoCS

Summary

- unless you do DRM, do not put secrets in binaries
- offload sensitive computations
  - smart cards, hardware security modules
- white-box cryptography is hard
  - we don't even know if it's actually possible
  - long history of failed attempts

Part 5: Homomorphic Cryptosystems

Why Homomorphic Crypto?

- inverse problem to DRM
- private data in the public cloud
  - reminder: cloud = someone else's computer
  - "someone else" has full control over execution
- how to do useful things without decrypting?

Homomorphism?

- $f(e(x), e(y)) = e(f(x, y))$
  - e is the encryption function
  - f is some useful operation
- example: f is multiplication, e is RSA
  - $x^k \cdot y^k \bmod m = (x \cdot y)^k \bmod m$
  - does not work for addition
- RSA is only partially homomorphic

Fully Homomorphic Encryption

- allows arbitrary computation
- needs unlimited addition and multiplication
  - the rest can be built from those
- first plausible system: Gentry's Cryptosystem
  - proposed in 2009
  - extremely slow: 30 minutes per 1 bit operation

Second Generation Systems

- based on the learning with errors problem
  - need to reconstruct a linear function
  - from a finite number of noisy samples
- AES-128 circuit as a benchmark
  - about 36 hours per block initially
  - down to 4 minutes by 2014
- amenable to SIMD-like evaluation
  - brings down AES-128 to 2s per block
  - by processing 120 blocks at once