# CS 526: Information Security

## Software Protection

# Software Protection: Framework

- In previous lectures on exploitation of software vulnerabilities: Remote attacker who wants to gain unauthorized access
  - Vulnerable software is a tool towards that goal
- In software protection framework: Attacker is not remote, has physical control of computing device (might already be root)
  - The software itself is the target of interest

# Attack scenarios

- Unauthorized software modification
  - Attacker wants to remove obstacles to piracy (e.g., license check, reliance on hardware token)
  - Attacker wants to add functionality (e.g., turn a demo version into a fully functional one)
  - Attacker wants to add malware functionality to the software
  - Attacker wants to cheat at online computer games (gain advantage over other players)

# Attack scenarios (cont'd)

- Malicious reverse engineering
  - Software contains valuable information that attacker wants to learn (e.g., trade secrets, military secrets, cryptographic keys, … )
- Code lifting
  - Software contains functionality that attacker wants to re-use in his own program
  - Attacker may not care to understand the internal details of the target functionality (only to make unauthorized use of it in another product)

# Attack scenarios (cont'd)

- Defeating digital rights management (DRM)
  - E.g., DRM media player is the only way to view encrypted media files, attacker wants the decrypted media files (might be easier than obtaining the crypto keys)
- Removing a watermark from the software
  - Watermark contains identity of attacker, who wants to remove it before unauthorized sharing (software piracy)

# Other examples of targets for attack

- Software running on cable TV set-top boxes
  – Attacker wants to view without paying
- Software running on taxi metering devices
  – Attacker wants to over-charge customers
- Software running on utility's smart meters
  – To under-pay, avoid rationing, cause blackout
- Software controlling a physical plant
  – Sabotage, terrorism

# Hardware Tokens – “Dongles”

- Plugged into the computer
  - Aim to thwart software piracy (massive copying)
  - Program requires plugged token’s presence
- Each program may require a separate token
  - “Dongle juggling”
- Low acceptance by users
- Defeatable
  - By analyzing traffic between dongle and program
  - Modify program, mimic dongle’s presence, …

# Smartcards

- Contain keys, capable of crypto computations
  – Resource-tight (limited storage, processing power)
  – Can be used to protect limited fragments of code and data (not large programs)
- Physically shielded from penetration
  – Card zeroes out its keys if it detects tampering
- Much more effort to attack than dongle
  – But possible (use physics, timing info, … )

# Secure co-processors

- Physically shielded from penetration
- Decrypt encrypted software, then run it
  - Keys are secure within co-processor
- Very difficult to defeat
- Drawbacks
  - Lower speed (heat dissipation issue)
  - Expense
  - Controversial (worry about hidden functionality)

# Encryption Wrappers

- Coarse grain wrappers
  - Whole program
- Fine grain wrappers
  - Function-level
  - Only for critical functions
- Defeatable
  - Because must decrypt before running

# Protection from Outside

- Change detection tools
  - E.g., Tripwire (protects other programs than itself on the system)
- Periodic detection
  - May be too late (not real-time)
- Attacker can disable
  - Attacker simply does not run Tripwire

# Protection from Outside: Signed code

- Unauthorized modification makes the program no longer match the signature, and it is then denied the right to execute
- OK if the attacker does not control the environment in which the program runs, but not if attacker has control of environment
  - E.g., attacker can skip signature-verification altogether

# Software that Self-Protects

- Detects unauthorized modification of itself, and reacts to it
- Possible reactions:
  - Notify (“call home”)
  - Self-repair
  - Crash
  - Display message (“shame on you!”)
  - More drastic actions (e.g., destroy the file system)

# Software that Self-Protects (cont'd)

- Can be effective in delaying software crackers
  - Some businesses happy with delay of a few weeks
  - Others need much longer delay of attackers
- Expensive and unwieldy if done manually
  - Must re-do manually after every legitimate modification to the software
- Better to automate adding the protection
  - Enables quick re-protection if needed

# Software that Self-Protects: Guards

- Automatically add, to existing software, protections (“guards”) that detect tampering with the software, and react to it
- Guards also protect each other
  - A network of mutual protection within software
- Guards are lightweight, stealthy, numerous, polymorphic
  - Adding guards does not modify the functionality of the software (unless tampering occurs)

# Guards (cont'd)

- Guards can protect implicitly or explicitly
  - Explicitly: Compute checksum of protected code fragment and compare to stored hash
  - Implicitly: Through deliberately introduced dependencies that are harmless unless tampering occurs
- Guard-installation is automated and is integrated with compilation
  - Software developers unaware of it all

# Protection Through Obfuscation

- To obfuscate a program = to transform it into a form that is more difficult for an adversary to understand or change than the original
- Source-level is expensive (in speed, program size), gives rise to compiler issues
- Low-level has less impact on speed, size
- Tools exist for automating the obfuscation process
  - Tools also exist for de-obfuscation

# Reverse Engineering: Data Flow Analysis

- Collect info about data in a program
- Static: Collect info without running program
- Dependency analysis
  - "On which other variables does x depend?"
  - "Where else will x be used later?"
- Determining structure of program
  - Main logical components
  - Interactions between them

# Data Flow Analysis (continued)

- Many tools exist
  - Useful for legitimate purposes, e.g.,
    - Fixing bugs in legacy systems
    - Conversion from one national currency to another, or to metric system
  - But the tools can also be used for attack
- De-obfuscation
  - Dead code elimination
  - Code simplification
  - “Constant folding”

# Making Data Flow Analysis More Difficult

- Use unstructured code
  - Unintelligible spaghetti of "go to" statements
- Aliasing
- Use intractable problems
  - Tautology (= Boolean expression that evaluates to 1; it is NP-complete to check whether tautology)
  - Prevents simplification
- Introduce new ("fake") dependencies
- Use data obfuscation

# Data obfuscation

- Storage
  - Local var => global var
- Encoding
  - x => f(x)  and, elsewhere, x by $f^{-1}(x)$
- Aggregation
  - 2-d array => 1-d array
- Ordering
  - A[j] => A[permutation(j)]

# Control Flow Obfuscation

- Transformation-based
- Hide real control flow
  - Change the grouping of instructions (aggregation)
  - Change ordering of instructions
  - Insert computations that have no net effect (adding a 0, multiplication by 1, use of math identities)
  - Create spurious program embedded into original (for obscuring important features of the original)
  - Opaque predicates (evaluate to true in a hard to recognize way)

# Control Flow Obfuscation: Cloning, Merging

1. Make copy P’ of program P
2. Modify P and P’ so they look very different
   – Re-name variables, mutate, obfuscate
3. Merge modified P and P’, getting P”
   – Within P”, introduce fake dependencies between P and P’ (so they cannot be disentangled)
4. Detect tampering by implicitly comparing values computed in P and P’

[ Repeat Steps 1-4 with P” playing the role of P ]