Software Security

Summary

Contents

Chapter 1

Language Based Security

1.1 Information Flow Security

1.1.1 Tracking Information Flow

Perl has a taint mode feature that allows the tracking of input. When active all forms of input to the programs are marked as "tainted". Tainted variables taint variables explicitly calculated from them and tainted data may not be used in any sensitive command (with some exceptions).

This mechanism implicits a set of security classes (tainted vs. untainted), as well as a classification of objects/information holders, a specification of when information can flow from onde security class to another and a way to determine security classes that safely represent the combination of two other.

1.1.2 Information Flow Policies

The goals of information security are confidentiality and integrity. Information flow policies specify how information should be allowed to flow between objects of each security class. To define one such policy we need:

- A set of security classes
- A can-flow relation between them
- An operator for combining them

Information Flow Policies For Confidentiality

Confidentiality classes determine who has the right to read and information can only flow towards confidentiality classes that are at least as secret.

Information that is derived from the combination of two security classes takes a confidentiality classes that are at least as secret as each of them.

Information Flow Policies For Integrity

Integrity classes determine who has the right to write and information can only flow towards integrity classes that are no more trustful.

Information that is derived from the combination of two integrity classes takes an integrity class that is no more trustful than each of them.

Formal Information Flow Policies

These policies can be described as a triple $(SC, \rightarrow, \oplus)$, where:

- SC is a set of security classes
- $\rightarrow \subseteq SC \times SC$ is a binary can-flow relation on SC
- $\oplus : SC \times SC \rightarrow SC$ is an associative and commutative binary class-combining or join operator on SC

Example high-low policy for confidentiality:

- $SC = \{H, L\}$
- $\rightarrow = \{(H, H), (L, L), (L, H)\}$
- $H \oplus H = H, L \oplus H = H, L \oplus L = L$

And for integrity:

- $SC = \{H, L\}$
- $\rightarrow = \{(H, H), (L, L), (H, L)\}$
- $H \oplus H = H, L \oplus H = L, L \oplus L = L$

Partial Order Policies

It often makes sense to assume that information can always flow within the same security level, security levels that are related to others in the same way are the same security level and, if information can flow from A to B and from B to C, it can flow from A to C. The flow relation $\rightarrow \subseteq SC \times SC$ is a partial order $(SC, \rightarrow)$ if it is:

- Reflexive: $\forall s \in SC, s \rightarrow s$
- Anti-symmetric: $s_1 \rightarrow s_2$ and $s_2 \rightarrow s_1$ implies $s_1 = s_2$
- Transitive: $s_1 \rightarrow s_2$ and $s_2 \rightarrow s_3$ implies $s_1 \rightarrow s_3$

When dealing with a partial order, the notation for $\rightarrow$ is $\leq$ and we can speak of security levels.

Hasse diagrams are convenient for representing information flow policies that are partial orders. They are directed graphs where security classes are nodes, the can-flow relation is represented by non-directed arrows, implicitly directed upward and reflexive/transitive edges are implicit.

1.1.3 Access Control to Information Flow Control

Information flow control focuses on how information is allowed to flow once an access control is granted. Access control is the control of interaction between subjects and objects, by validating access rights of subjects to resources of the system.

Discretionary Access Control (DAC)

Restricts access based on the identity of subjects and a set of access permissions that can be determined by subjects.

It has a limitation where access permissions might allow programs to, in effect, circumvent the policies. This can be done legally by means of information flows that are encoded in the program, or illegally, when vulnerabilities in programs and language implementations can be exploited by attackers.

Mandatory Access Control (MAC)

Restricts access based on security levels of subjects (their clearances) and objects (their sensitivity). Controls how information flows in a system based on whom is performing each access. It has limitations of restrictiveness and covert channels.

1.1.4 Encoding and Exploiting Information Flows

Objects may be classified as follows:

- Object - resource holding or transmitting information
- Security class/label - specifies who can access objects of that class
- Security labelling - assigns security classes to objects (statically or dynamically)

We use a standard imperative language where information containers are variables, where X_L denotes that a variable X has security level L. The information flow policy is as follows:

We want to ensure that propagation of information by programs respects information flow policies, i.e. there are no illegal flows. This means an attacker cannot infer secret input or affect critical output by inserting inputs into the system and observing its outputs.

1.2 Noninterference

1.2.1 Definition

A program is secure if, for every observational level L, for any two runs of the program that are given the same low inputs, if the program terminates in both cases, then it produces the same low outputs.

1.2.2 Attackers

Concurrent Attacker

An attacker program that is concurrently composed with the observed program does not depend on its termination. It has access to "low" outputs, and possibly non-termination (or even intermediate steps). Considering the following programs:

- p_L="$file_L$"; if RUID access to p_L then f=open(p_L);$f = 0$ (has RUID=L and EUID=H)
- $p_L = file_H$

Both are safe according to our notion of noninterference, but when composed concurrently, the program is insecure.

Possibilistic Input-Output Noninterference: is sensitive to whether the program is capable of terminating and producing certain final outputs.

Intermediate-Step Attacker

- $x_L := y : H; x_L := 1$

Possible low outcomes do not depend on y_H. However, the intermediate steps differ.

Intermediate-step-sensitive Noninterference: is sensitive to intermediate steps of computations.

Time-Sensitive Attacker

- $x_L := 0$; if y_H then skip else skip;skip;skip;skip ; $x_L := 1$

Possible outcomes and intermediate steps do not depend on y_H. However, the time it takes to change the value of x_L is different.

Temporal Noninterference: is sensitive to the time it takes to produce outputs.

Probabilistic Attacker

- $x_L := y_H \mid\mid x_L :=$ random(100)

Possible outcomes do not depend on y_H. However, the probability of the value of x_L revealing that of y_H is higher.

Probabilistic Noninterference: is sensitive to the likelihood of outputs.

1.2.3 Downgrading

Noninterference is simple and provides strong security guarantees. But sometimes we need to leak information in a controlled way.

- Declassification (for confidentiality) Example: flow declarations locally enable more flows

```
declassify password:L in
   if (password_H == attempt_L) {
     then print_L "Right!";
     else print_L "Wrong!";
   }
```

- Endorsement (for integrity) Example: pattern matching in Perl's taint mode

```
if ($filename_L =~ /^([-\@\w.]+)$/) {
  $filename_H = $1_H;
  open_H(FOO,"> $filename_H");
}
```

1.3 Formal Semantics

We will use two techniques to define the semantics of a programming language:

- Denotational semantics for expressions: defines mathematically what is the result of a computation.
- Operational semantics for instructions: describes how the effect of a computation is produced when executed on a machine

1.3.1 WHILE Language

This language has the following syntatic categories:

- c: constants
- x: variables
- a: arithmetic expressions
- t: tests
- S: statements

And follows the grammar:

- Operations: $op ::= + | - | \times | /$

- Comparisons: $cmp ::= < \mid \leq \mid = \mid \neq \mid \geq \mid >$
- Expressions: $a ::= c|x|a_1\ op\ a_2$
- Tests: $t ::= a_1\ cmp\ a_2$
- Statements: $S ::= x := a|skip|S_1; S_2|if\ t\ then\ S\ else\ S|while\ t\ do\ S$

The state/memory is represented as a function that maps variables to integers, ρ, for example, $\rho(x) = 1$. Now, we define the following semantic functions:

- $\mathcal{A}$: function that maps pairs of arithmetic expression and state to integers ($\mathcal{A}(x)_\rho = \rho(x)$)
- $\mathcal{B}$: function that maps pairs of test and state, to booleans ($\mathcal{B}(a_1\ cmp\ a_2)_\rho = \mathcal{A}(a_1)_\rho\ cmp\ \mathcal{A}(a_2)_\rho$)
- $\mathcal{S}$: partial function that maps pairs of statement and state to state ($< S, \rho > \rightarrow \rho'$: when executing program $\mathcal{S}$ on memory ρ we obtain the new memory ρ'. $< S, \rho > \rightarrow < \mathcal{S}', \rho' >$: Performing one step of program $\mathcal{S}$ on memory ρ leaves the continuation $\mathcal{S}'$ and produces new memory ρ')

The list of big-step axioms and rules is as follows:

Assignment: $< x{:=}a, \rho > \rightarrow \rho[x \mapsto \mathcal{A}[\![a]\!]\rho]$

Skip: $< skip, \rho > \rightarrow \rho$

Sequential composition: $$\frac{< S_1, \rho > \rightarrow \rho' \quad < S_2, \rho' > \rightarrow \rho''}{< S_1; S_2, \rho > \rightarrow \rho''}$$

Conditional test: $$\frac{< S_1, \rho > \rightarrow \rho'}{< \text{if t then } S_1 \text{ else } S_2, \rho > \rightarrow \rho'} \text{ if } \mathcal{B}[\![t]\!]\rho = true$$

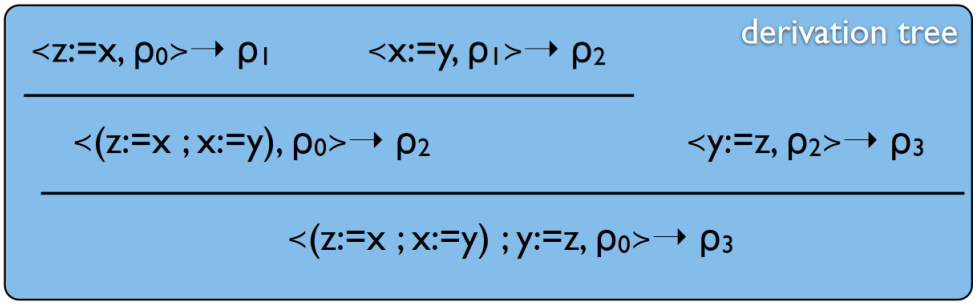
$$\frac{< S_2, \rho > \rightarrow \rho'}{< \text{if t then } S_1 \text{ else } S_2, \rho > \rightarrow \rho'} \text{ if } \mathcal{B}[\![t]\!]\rho = false$$

While loop: $$\frac{< S, \rho > \rightarrow \rho' \quad < \text{while t do S}, \rho' > \rightarrow \rho''}{< \text{while t do S}, \rho > \rightarrow \rho''} \text{ if } \mathcal{B}[\![t]\!]\rho = true$$

$$< \text{while t do S}, \rho > \rightarrow \rho \quad \text{if } \mathcal{B}[\![t]\!]\rho = false$$

For example, the sequential composition rule could be read as follows: When the first program starting on ρ produces ρ' and the second program starting on ρ' produces ρ", then the entire sequential composition starting on ρ produces ρ". The following is an example of a chaining of these rules:

- Evaluate (z:=x ; x:=y) ; y:=z, starting from a state ρ_0 that maps all variables except x and y to 0, and has $\rho_0(x) = 5$ and $\rho_0(y) = 7$.



$\rho_1 = \rho_0[z \mapsto 5]$ $\rho_2 = \rho_1[x \mapsto 7]$ $\rho_3 = \rho_2[y \mapsto 5]$

1.4 Security Properties and Enforcement Mechanisms

The definition of a security property is often not enough, since devs make mistakes and understanding whether a program satisfies the property is not always straightforward. At their core, security properties are about behaviour:

- Functional correctness
- Robustness
- Safety
- ...

Enforcement mechanisms were created to this end, automating the algorithm of preventing any given program from performing "unwanted" behaviors.

1.4.1 Program Analysis

Program analysis is the process of automatically analyzing the behavior of computer programs. The main aims of program analysis are:

- Optimization - about performance, to compute in a more efficient way
- Correctness - about assurance, to compute as intended

Automatic analysis can give stronger guarantees in less time, but is limited in scope and precision, which may come in the form of false positives or negatives.

Security properties typically talk about behavior of programs, and are often undecidable. On the other hand, enforcement mechanisms provide an automatic way of accepting/rejecting the behavior of programs, and are expected to be decidable.

Timing of Program Analysis

Program analysis may be done before program execution (static), during execution (dynamic) or using a combination of both, using the output of one to another (hybrid).

1.4.2 Static Analysis Mechanisms

Static analysis is closely related to compilation. Some tools used for static analysis include:

- String matcher: runs directly over source code. Simple tools like grep and findstr can do a very basic form of analysis.
- Lexical analyzer: runs over the tokens generated by the scanner. Can look for dangerous library/system calls.
- Semantic analyzers:
  - Control flow analysis: performs checks based on the possible control paths of a program; used to verify properties that depend on the sequencing of instructions.
  - Data-flow analysis: gathers information about the possible set of values calculated at various points of a program.Can determine where an actual value assigned to a variable might propagate.
  - Type checking: associate types to selected programs that fulfill certain requirements (eg. are considered correct with respect to a property)

Interactive Analysis

Verification of complex properties can be achieved with more human intervention, such as model checking or program verification.

- Model checking: checks a model (description) of a program, or the code itself. Enables to check its design.
- Program Verification: formally proves a property about a program. Uses a specification language (program logic) for expressing properties of a program and an associated logic for (dis)proving that programs meet specifications.

Static analysis for Information Flow

1. Definition of the language
2. Information flow policy of security levels
3. Classification of objects into security levels
4. Security property of programs
5. Mechanism for selecting secure programs
6. Guarantees about the mechanism

Chapter 2

Vulnerabilites And Secure Software Design

In secure software design, there are 3 main security attributes: confidentiality, integrity and avaliability.

2.1 Vulnerabilities

A vulnerability is a system defect relevant security-wise, which may be exploited by an attacker to subvert security policy. Vulnerabilities may be classified as:

- Design vulnerabilities
- Coding vulnerabilities
- Operational vulnerabilities

2.2 Attacks

Attacks enter through interfaces, the attack surfaces. Attacks can be techincal or through social engineering, directed or not, manual or automated.

2.2.1 Manual Attacks

Some examples of manual attacks include:

- Footprinting
- Scanning
- Enumeration
- Discovering vulnerabilites
- ...

2.2.2 Automated Attacks

Worm

A worm is composed of a target selector, a scanning motor, a warhead (exploit code), a load and a propagation motor.

Drive-by Download

Performed by web pages with malware. When user accesses one with a vulnerable browser, the malware exploits the vulnerability.

Viruses and Trojans

Viruses are similar to worms but propagate with physical contact (usb drives, disks, ...). Trojans are also similar but requires the user to run an infected program (e.g. emails with attachments).

2.2.3 Torpig

Torpig is a sophisticated malware. It infects bots with drive-by download. Attackers modify legitimate but vulnerable server for some webpages to request JavaScript code from the attacker's web server:

1 The victim's browser accesses the vulnerable server

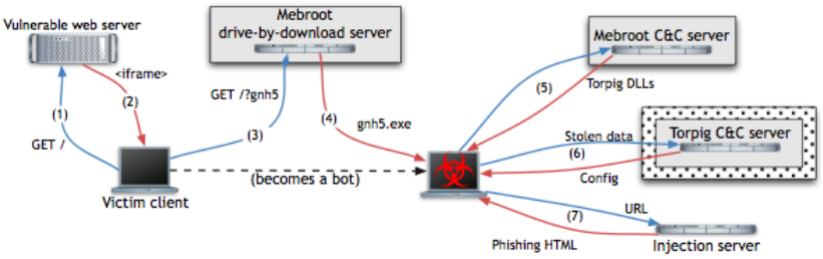
2 JavaScript code exploits the browser/plugins/etc.

3-4 If an exploit is successful, the script downloads and installs the Mebroot rootkit (replaces Master Boot Record) – victim becomes a bot. Mebroot has no attack capacity.

5 Contacts C&C server to obtain malicious modules and stores them encrypted in directory system32 and changes the names and timestamps to avoid suspicions.
Every 2h contacts C&C server: sends its configuration (type/version of modules); gets updates; communication is encrypted over HTTP

6 Every 20 minutes contacts C&C to upload stolen data

7 When victim visits domain from a list (e.g., a bank), the bot contacts an injection server. Injection server returns attack data: URL of trigger page in the legitimate domain (typ. the login page), where to send results, etc. When user visits trigger page, Torpig asks injection server for another page (e.g., that asks for credit card number)



2.3 Protection in Operating Systems

Protection is emplyed to ensure that objects are not accessed by unauthorized subjects. There are two aspects: separation and mediation.

2.3.1 Separation

Common operating systems (Unix, Windows) run software basically in two modes, enforced by the CPU:

- Kernel mode: software can play with any system resource (memory, I/O devices,...)
- User mode: access to resources is controlled by the OS. Software has to call the OS kernel to make privileged operations

There are several forms of separation:

- Physical separation: different processes use different devices (e.g. printers for different levels of security)
- Temporal separation: processes with different security requirements are executed at different times
- Logical separation: processes operate under the illusion than no other processes exist
- Cryptographic separation: processes use cryptography to conceal their data and/ or computations in a way that they become unintelligible to other processes

Memory Protection

Logical separation is often used to achieve memory protection. The most common solutions are segmentation (program is split in pieces with logical unit, such as code, data, stack...) and paging (program is divided in pages of the same size).

From a protection point of view, pages are similar to segments: a process can only access a segment only if appears in its segment translation table or can only see a page if it appears on its table.

2.3.2 Access Control

Access control is concerned with validating the access rights of subjects to resources of the system. It should be implemented by a reference monitor, following 3 principles:

- Completeness: it must be impossible to bypass
- Isolation: it must be tamperproof
- Verifiability: it must be shown to be properly implemented

Some basic access control mechanisms include:

- Access control lists (ACLs): Each object is associated with a list of pairs (subject, rights)
- Capabilities: Each subject has a list of objects that it may access, i.e. pairs (object, rights). Capabilities are cryptographically protected against modification and forging
- Access control matrix: A matrix with lines per subject, columns per object, rights in the cells

There are also two basic access control policies:

- Discretionary Access Control (DAC): access policy defined by the user
- Mandatory Access Control (MAC): access policy defined by an administrator

Unix Access Control

Each user in Unix has a username and a user ID (UID). Users can also belong to one or more groups, each with a group ID (GID). Objects (such as files or directories) have an owner identified by a real UID and a real GID. Access permissions are set for the owner, group, and others (world), with read (r), write (w), and execute (x) permissions.

When processes interact with objects, their effective UID and effective GID are used to determine access permissions. The kernel compares the effective UID and GID with the permissions of the object to decide whether to grant or deny access.

The setuid and setgid bits are additional permission bits that can be set on executable files. When the setuid bit is set on an executable file, the program is executed with the effective UID of the file's owner. Similarly, when the setgid bit is set, the program is executed with the effective GID of the file's group owner. Programs with setuid and owner UID 0 (root) are potential targets for privilege escalation attacks because they may execute with elevated privileges, providing an opportunity for attackers to exploit vulnerabilities.

2.4 Race Conditions

Race conditions violate the assumption of atomicity. The vulnerability lies in a problem of concurrency or lack of proper synchronization. There are several sources of races:

- Shared data (files and memory)
- Preemptive routines (signal handlers)
- Multi-threaded programs

There are 3 main kinds of race conditions, discussed next.

2.4.1 TOCTOU

TOCTOU stands for "time-of-check to time-of-use". It is often perpetrated through the use of symbolic links (special files that reference other files/folders).

The *access* function is specially vulnerable as it was designed for *setuid* programs. It does privilege check using the process' real UID instead of the effective UID.

When a call with a pathname is done ($open, access, stat, lstat, ...$), the pathname is resolved until the inode is found, so if two calls are made one after the other the path can lead to different inodes. The solution is to avoid the two sequential resolutions by avoiding using filenames inside the program. Functions such as $fstat, fchmod$ and $fchown$ are safe.

2.4.2 Temporary Files

Temporary files have the added problem of being in a shared directory. The typical attack is as follows:

- Privileged program checks that there is no file X in /tmp
- Attacker races to create a link called X to some file, say/etc/passwd
- Privileged program attempts to create X and opens the attacker's file doing something undesirable that its privileges allow

One may use the *mkstemp* function to create a unique, currently unused, filename from template (*mkdtemp* for directories).

Possible solutions include setting *unmask* appropriately and using $fopen$ instead of *open*.

2.4.3 Concurrency and Reentrant Functions

In the previous cases, concurrency is created by the attacker, with malicious intention. In many cases in which there is normal concurrency of operations on objects, operations may have to be executed atomically (using mutual exclusion mechanisms).

A function is reentrant if it works correctly even if its thread is interrupted by another thread that calls the same function. Such functions cannot use static variables, global variables, other shared resources like libraries (i.e and can only call other reentrant functions.

In some cases, unix signals may be useful in indicating asynchronous events to a process.