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

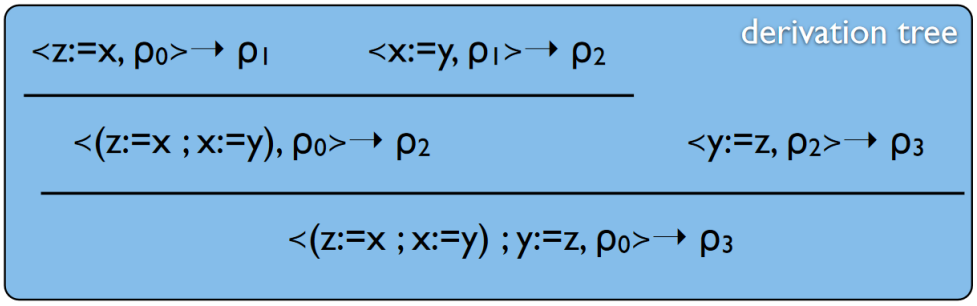
$$\frac{< S_2, \rho > \rightarrow \rho'}{< \text{if t then } S_1 \text{ else } S_2, \rho > \rightarrow \rho'} \text{ if } \mathcal{B}[\![t]\!]\rho = false$$

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 $\rho_0(y) = 7$.



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

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

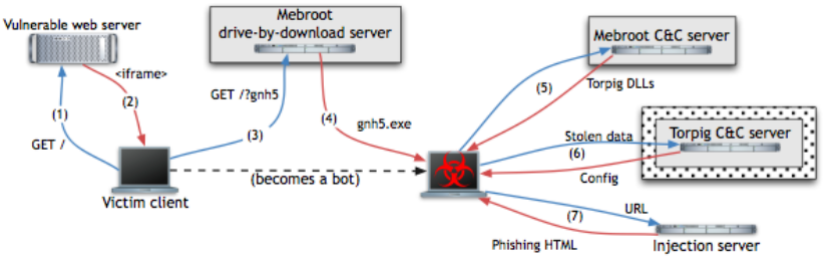
2 JavaScript code exploits the browser/plugins/etc.

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)


Vulnerable web server
<iframe>
(1)
GET /
(2)
Victim client
Mebroot
drive-by-download server
GET /?gnh5
(3)
(4)
gnh5.exe
(becomes a bot)
Mebroot C&C server
(5)
Torpig DLLs
Torpig C&C server
Stolen data
(6)
Config
URL
(7)
Phishing HTML
Injection server