Blare

policy-based intrusion detection systems

Guillaume Brogi
www.blare-ids.org

2nd January 2014
Outline

General context and background

Information flow tracking

OS level detection
   Global overview
   Implementation

Distributed Detection

Demo

Conclusion
Traditional IT security context

Security mechanisms to prevent intrusions

- Access control and authentication
- Static analysis and testing
- Avoiding the execution of arbitrary code: Address Space Layout Randomization (ASLR), NX bit, PAX (Linux)
- Firewalls

Security flaws remain

- IT systems have increasing complexity and keep changing
- It is difficult to maintain and enforce the overall security policy
- Intrusions occur
- Intrusion Detection Systems (IDSs) have become a necessary addition to most IT infrastructures
Intrusions

- Web attacks (cross site scripting, upload exploits, etc. . . )
- Malware, rootkits
- SQL injections
- Botnets
- etc. . .

Common characteristics

- Some information is modified by an unauthorized user or application
- Some information leaks towards an unauthorized user or application
Approaches

- **Misuse detection**: detect intrusion based on the knowledge of abnormal behaviour (signatures) e.g. Snort, Prelude etc...
- **Anomaly detection**: detect intrusions based on the knowledge of normal behaviour.
General context and background

Information flow tracking

OS level detection
  Global overview
  Implementation

Distributed Detection

Demo

Conclusion
Information flow tracking

Principle

- Objects of the system (or inside supervised programs) are considered as containers of information (i.e. files, variables etc...).
- We attach labels called tags to such containers.
Taint marking

**Tags**

- Contains taint information (meta-information) describing content.
- Help us track information flows (by propagating taint data).

**Information flow policy**

- It defines the legal content of containers.
- Protects confidentiality and integrity.
Java programs
JBlare, tracks information flows within Java programs by running a modified Java Virtual Machine (JVM).

Linux Kernel
KBlare, tracks information flow between operating system objects.

Android
AndroBlare, tracks information flow between Android applications (malware detection).
Outline

- General context and background
- Information flow tracking
- **OS level detection**
  - Global overview
  - Implementation
- Distributed Detection
- Demo
- Conclusion
Dynamic information flow tracking

- We track information flows at runtime within the **operating system**, at the kernel level.
- We attach **tags** to operating containers of information: **files**, **processes**, **sockets**, memory pages, shared memory mappings, pipes, message queues, **etc.**
Tainting information at the OS level

Information tags

- Contain *meta-information*, describing content.
- It is updated after each information flow so as to describe the new content.
- It traces the origin of the content.
Information tags

- Contain *meta-information*, describing content.
- It is updated after each information flow so as to describe the new content.
- It traces the origin of the content.
Tainting information at the OS level

Information tags

- Contain *meta-information*, describing content.
- It is updated after each information flow so as to describe the new content.
- It traces the origin of the content.
Information tags

- Contain *meta-information*, describing content.
- It is updated after each information flow so as to describe the new content.
- It traces the origin of the content.
Tainting information at the OS level

Policy tags

- Contain *meta-information*, describing the legal content of the containers.
- It is set beforehand and doesn’t change.
- It is checked after each information flow and raises an alert if the new content is not legal.
An information flow towards a container is legal iif the updated information tag (after the flow occurred) is a subset of one of the sets of its policy tag:

\[ itag(c) \subseteq ptag(c) \iff \exists A \in ptag(c) | itag(c) \subseteq A \]
An information flow towards a container is legal iif the updated information tag (after the flow occurred) is a subset of one of the sets of its policy tag:

\[ itag(c) \subseteq ptag(c) \iff \exists A \in ptag(c) | itag(c) \subseteq A \]
An information flow towards a container is legal if the updated information tag (after the flow occurred) is a subset of one of the sets of its policy tag:

\[ itag(c) \subseteq ptag(c) \iff \exists A \in ptag(c) \mid itag(c) \subseteq A \]
An information flow towards a container is legal iff the updated information tag (after the flow occurred) is a subset of one of the sets of its policy tag:

\[ itag(c) \subseteq ptag(c) \iff \exists A \in ptag(c) | itag(c) \subseteq A \]
An information flow towards a container is legal if the updated information tag (after the flow occurred) is a subset of one of the sets of its policy tag:

\[ itag(c) \leq ptag(c) \iff \exists A \in ptag(c) \mid itag(c) \subseteq A \]
An information flow towards a container is legal if the updated information tag (after the flow occurred) is a subset of one of the sets of its policy tag:

\[ \text{itag}(c) \subseteq \text{ptag}(c) \iff \exists A \in \text{ptag}(c) \mid \text{itag}(c) \subseteq A \]
An information flow towards a container is legal if and only if the updated information tag (after the flow occurred) is a subset of one of the sets of its policy tag:

\[
itag(c) \subseteq ptag(c) \iff \exists A \in ptag(c) \mid itag(c) \subseteq A
\]
LSM framework

- Developed by the NSA for SELinux
- A number of hooks (used for MAC) are available
- We infer information flows from there (we added a couple of hooks (e.g. `shmdt`, `munmap`).

Blare IDS

www.blare-ids.org
**Linux kernel**

- Linux kernel ≈ 15 000 000 lines of code
- 171 LSM hooks

**Blare**

- 6877 lines, 39 files in `security/blare`
- 1 file (23 lines) added: `Documentation/security/blare.txt`
- 16 lines added in 3 files so that is seen as a LSM and compiled with the kernel
- 28 lines in 3 files to add the `shmdt` hook
Outline

General context and background

Information flow tracking

OS level detection
  Global overview
  Implementation

Distributed Detection

Demo

Conclusion
Tainting information in distributed systems

Labeling network packets

- We need to taint outgoing traffic: CIPSO (Commercial IP Security Option) labels.
- The remote hosts need to know how to interpret it: distributed policy.

Solutions

1. Using a bitmap (240 bits = 240 tags): each host of the group has a slot in the 240 bits to represent its tags ($I_k$).
2. Using a distributed security token management system: each token is a unique ID representing one itag.
Outline

General context and background

Information flow tracking

OS level detection
  Global overview
  Implementation

Distributed Detection

Demo

Conclusion
Setup

<table>
<thead>
<tr>
<th>files</th>
<th>tags</th>
</tr>
</thead>
<tbody>
<tr>
<td>vhost1</td>
<td>1</td>
</tr>
<tr>
<td>vhost2</td>
<td>2</td>
</tr>
<tr>
<td>apache2</td>
<td>3</td>
</tr>
<tr>
<td>libphp.so</td>
<td>4</td>
</tr>
<tr>
<td>DB1: wp_users</td>
<td>5</td>
</tr>
<tr>
<td>DB1: wp_posts</td>
<td>6</td>
</tr>
<tr>
<td>DB2: wp_users</td>
<td>7</td>
</tr>
<tr>
<td>DB2: wp_posts</td>
<td>8</td>
</tr>
<tr>
<td>postgres</td>
<td>9</td>
</tr>
</tbody>
</table>

Policy for apache2 and postgres:
\{-9, -4, -3, 1, 4, 5, 6\}\{-9, -4, -3, 2, 4, 7, 8\}
Debian GNU/Linux 6.0 apache ttyS0
apache login:

Debian GNU/Linux 6.0 postgres ttyS0
postgres login:

```
%   *
+   *
```

Easy phishing: Set up email templates, landing pages and listeners in Metasploit Pro’s wizard -- type 'go_pro' to launch it now.

```
= [ metasploit v4.8.2-2014011501 [core:4.8 api:1.0] ]
+ -- ---[ 1249 exploits - 681 auxiliary - 201 post ]
+ -- ---[ 328 payloads - 32 encoders - 8 nops]
msf > []
```
| Frame 4932: 95 bytes on wire (760 bits), 95 bytes captured (760 bits) on interface 0 |
|Internet Protocol Version 4, Src: 10.0.51.52 (10.0.51.52), Dst: 10.0.51.51 (10.0.51.51) |

- **Header length**: 40 bytes
- **Differentiated Services Field**: 0x00 (DSCP 0x00: Default; ECN: 0x00: Not-ECT (Not ECN-Capable Transport))
- **Total Length**: 81
- **Identification**: 0xbf98 (49048)
- **Flags**: 0x02 (Don't Fragment)
- **Fragment offset**: 0
- **Time to live**: 64
- **Protocol**: TCP (6)
- **Header checksum**: 0x58dc [correct]
- **Source**: 10.0.51.52 (10.0.51.52)
- **Destination**: 10.0.51.51 (10.0.51.51)

**Options**: (28 bytes), Commercial Security, End of Options List (EOL)

- **Type**: 134
- **Length**: 26
- **DOI**: 3
  - **Tag Type**: Restrictive Category Bitmap (1)
  - **Sensitivity Level**: 200
  - **Categories**: 110, 115, 116, 120, 121, 123

End of Options List (EOL)

- **Type**: 0

**Transmission Control Protocol, Src Port**: postgresql (5432), **Dst Port**: 41365 (41365), **Seq**: 1, **Ack**: 9, **Len**: 1

```
0000 52 54 92 bd 9a 1e 52 54 b5 a4 84 76 08 00 4c 00 RT...RT ...v..L.
0010 00 51 bf 98 40 00 40 06 58 dc 0a 00 33 34 0a 00 .Q...8. X...34...
0020 33 33 86 1a 00 00 00 03 01 14 00 c8 00 00 00 00 33 ............
0030 00 00 00 00 00 00 00 00 00 00 02 13 dc 00 00 15 38 ............8
0040 a1 95 02 b6 56 70 5a 09 c4 84 80 18 03 89 96 93 ...VpZ. ........
0050 00 00 01 01 08 0a 00 01 e9 e8 00 01 f6 c3 53 ............5
```
Outline

General context and background

Information flow tracking

OS level detection
  Global overview
  Implementation

Distributed Detection

Demo

Conclusion
Conclusion

Advantages

▶ Being able to detect which information has leaked.
▶ Locating infected and/or corrupted data.
▶ Determining the targeted hosts.

Limitations

▶ We cannot see what happens “inside” processes from the OS level.
▶ It works well when processes have short execution times (forks).
▶ It is very bad with buses (DBUS etc...) which tend to taint everything around → false positives.
▶ Excluding buses leads to potential evasion issues.
Conclusion

Future work

**Instrumentation of code**

- Specific applications leading to a great number of false positives may be instrumented.

**Combining OS level + application level detection**

- OS level monitor + JVM etc.
- Defining a protocol (probably on top of Netlink) between the monitors
- Automatic instrumentation of C code (Frama-C plugin) by another Ph.D. student of the CIDRE team.
Thank you for listening

www.blare-ids.org