A taint marking approach to confidentiality violation detection

Christophe Hauser[1,2], Frédéric Tronel[1], Jason Reid[2], Colin Fidge[2]

[1] Supélec
firstname.lastname@supelec.fr

[2] Queensland University of Technology
jf.reid@qut.edu.au, c.fidge@qut.edu.au

Abstract

This article presents a novel approach to confidentiality violation detection based on taint marking. Information flows are dynamically tracked between applications and objects of the operating system such as files, processes and sockets. A confidentiality policy is defined by labelling sensitive information and defining which information may leave the local system through network exchanges. Furthermore, per application profiles can be defined to restrict the sets of information each application may access and/or send through the network. In previous works, we focused on the use of mandatory access control mechanisms for information flow tracking. In this current work, we have extended the previous information flow model to track network exchanges, and we are able to define a policy attached to network sockets. We show an example application of this extension in the context of a compromised web browser: our implementation detects a confidentiality violation when the browser attempts to leak private information to a remote host over the network.

1 Introduction

Over the past decade, firewalls and antivirus software have become a necessary addition to the security components of operating systems, including those of mobile phones and embedded devices. More recently, sandboxing and access control tools such as AppArmor (Novell/SUSE n.d.) and SELinux (Stephen Smalley 2002) have also emerged, aiming to protect the operating system from untrusted applications. However, even if such components successfully protect the applications and the operating system, in practice they do not guarantee the protection of confidential data or users' privacy. The reason for this is that existing solutions either focus on mandatory access control resulting in a lack of flexibility, or enforce data flow policies at the network or transport level without any knowledge of the actual content. With the growing number of untrusted applications installed on portable computing devices such as smartphones, and the prevalence of untrusted scripts from remote services executed locally by web browsers, potential sensitive data leaks are becoming one of the most important threats for end users. For instance, a malicious script executed by a web browser may illegally access sensitive information and send it through the network, while the current user may legally access it manually with the same application. Current approaches are not sufficient to deal with such situations as they cannot catch indirect information flows (Stéphane Geller, Christophe Hauser, Frédéric Tronel, Valérie Viet Triem Tong 2011) (e.g., an information flow from the content of a sensitive file towards a network socket opened by the web browser). In this work, we present a novel approach to confidentiality violation detection based on dynamic taint marking. We extended and implemented Blare, a model of information flow tracking at the operating system level. Our extension is based on a *network policy*, stating how information is allowed to leave the local system through network exchanges and which applications may do so. This article is organised as follows: we first present existing work in the literature related to operating system level security, mandatory access control, firewalls and deep packet inspection in Section 3. Then, we present our approach of confidentiality violation detection based on taint marking and its implementation in the Linux kernel in Section 4. In Section 5.1, we report experimental results that demonstrate the effectiveness and efficiency of the proposed approach in detecting a leakage of sensitive data through a web browser running untrusted scripts. Finally, in Section 6, we discuss these results and possible future improvements.

2 Approach overview

We have developed a framework that allows users to trace how their private data is used by applications, and to monitor sensitive information that flows out over the network. Most of today's personal computers rely on untrusted third party applications such as browser plugins or so called 'apps'. Many of these are closed source, which makes static analysis impossible (in the case of native code). And even in the case of opensource applications, there is always a risk of security flaws or coding errors potentially leaking sensitive data. Dynamically detecting the leak of sensitive information is challenging given that:

- One application can exchange information with another using IPC, shared memory, etc.
- It is impractical to modify off-the-shelf applications; instead, we prefer to implement a reference monitor in the operating system kernel as a more pragmatic solution.
- The performance overhead must be small to maintain a responsive system, i.e., not affecting the user's experience and causing them to disable the security mechanisms.

We use dynamic tracking of information flows between objects of the operating system in order to monitor sensitive data leaks. A defining aspect of our approach is that we distinguish *data* from *containers*: data is the actual information we track, whereas containers are storage entities such as files, memory pages, etc. Sensitive data is first identified and their containers are labeled with meta-data called *tags*. As information flows between containers, tags are dynamically updated to reflect the container's content.

When it comes to protecting sensitive data against leakage by untrusted applications or via malware that exploits security flaws, existing approaches have several limitations. Individuals can use software firewalls on their internet-connected personal/portable computing devices to filter network connections without changing the security policy of the underlying operating system. However, while such mechanisms may successfully protect a host from outside threats, they typically do not prevent the leak of information by untrusted or misconfigured applications. Deep packet inspection firewalls are able to identify data patterns in network packets, however this approach is too coarse-grained to efficiently track the presence of sensitive data in network exchanges and is thus not an effective solution to protect against sensitive data leaks.

Mandatory access control tools such as AppArmor (Novell/SUSE n.d.) and Tomoyo (Harada et al. 2003) are similarly not practical when it comes to protecting confidentiality:

- When used in enforcement mode, information flows are blocked, which may break some functionalities. This effectively renders the approach unusable for most end users.
- When used in permissive mode, these tools are unable to track indirect information flows (Stéphane Geller, Christophe Hauser, Frédéric Tronel, Valérie Viet Triem Tong 2011).

Figure 1 presents our approach to taint tracking for monitoring data leaks. A kernel reference monitor has been implemented in the Linux Kernel and allows for efficient dynamic information flow tracking at the level of system objects (processes, filesystem inodes, etc.).

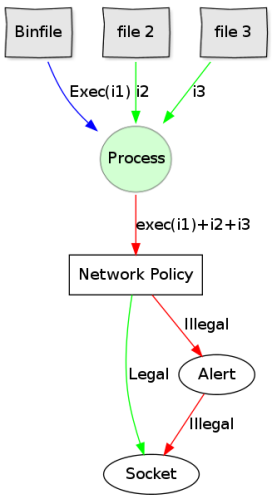

Figure 1: Network information flow tracking

Sensitive data is labelled at the filesystem level, and the level of granularity of our approach is at the file level (i.e., files are considered as atomic pieces of information). Our implementation takes advantage of the Linux Security Modules (LSM) framework available in the Linux kernel, and taint propagation is triggered by access control hooks. Our design goals are to provide a model that is easy to use, does not lock all the system by default by labelling only the sensitive information, and does not miss any information flow (no false negatives). We consider false positives as an acceptable fact in most situations where sensitive data is involved and should leak by no means.

3 Background

This section provides an overview of existing solutions such as firewalls, deep packet inspection, traditional operating system security mechanisms, access control and host-based intrusion detection systems. It also highlights the deficiencies in these approaches in addressing the data leakage problem examined in this paper.

3.1 Firewalls

Firewalls are devices or software that filter network traffic at different layers of the ISO network model. They can be set up to restrict access to a personal machine or a company's network from other untrusted networks, thus creating trust boundaries (Ingham & Forrest 2002). Individuals can use software firewalls on their personal/portable computers to define and enforce policies concerning both incoming and outgoing network traffic. Deep Packet Inspection (DPI) firewalls identify anomalous patterns in traffic volumes by inspecting both the headers and content of packets. They provide the capability of identifying anomalous network traffic as well as managing normal traffic. They also form the core of many commercially-available firewalls and intrusion detection systems (IDS). Tamer et al. (AbuHmed et al. 2008) present a survey of the Deep Packet Inspection algorithms, implementation techniques, research challenges and their usage in several existing technologies for intrusion detection systems. Some of the highlighted challenges include the complexity of research algorithms, the ever-increasing number of attack signatures (which negatively impacts on performance) and the increasing prevalence of encrypted data which DPI cannot examine. In terms of the problem this paper seeks to address, a key drawback of DPI is that the sensitive data must first be exhaustively enumerated in signatures and this may be difficult for non-technical users.

3.2 Access control

Discretionary access control (DAC) is the most commonly used access control model and is the default on UNIX based systems. Access is restricted given the *identity* and the *group* of the subject trying to access an object. While traditional *discretionary access control* lets subjects transfer certain permissions to each other at their own discretion and remains widely used, previous research on *mandatory access control* (MAC) has led to implementations in common operating systems, such as Linux, FreeBSD, Mac OS X and Windows. Linux and FreeBSD have been extended with generic access control frameworks: the Linux Security Modules (LSM) (Wright et al. 2002) and TrustedBSD (Watson & Vance 2003). These

frameworks provide sets of hooks for mediating access to resources (files, sockets, IPC, etc.). So-called LSM modules can implement various models and policies within the Linux kernel by using the LSM Framework. SELinux (Stephen Smalley 2002) emerged from research led by the National Security Agency. It is the first security module available in Linux, and it has been designed to implement a flexible MAC mechanism called *domain and type enforcement* (DTE). Other LSM modules include AppArmor (Novell/-SUSE n.d.), Smack (Schaufler n.d.) and Tomoyo (Harada et al. 2003). AppArmor (Novell/SUSE n.d.) is a MAC implementation available in the Linux kernel which aims to be a simpler alternative to SELinux. It is used by default by Novell in their products and comes with a predefined policy, and a set of generic definitions to ease the difficulty of creating new policies. When in use, LSM modules block illegal accesses to resources before they can occur. This is sometimes referred to as *enforcement mode*. Most of these modules also support a *permissive mode*, in which illegal accesses are logged but not blocked. Such behaviour is comparable to policy-based IDSs (Georges et al. 2009). By using MAC mechanisms, one can finely control the operations each subject is allowed to perform on the objects of the system. When configured correctly, those mechanisms can significantly improve security by rejecting illegal accesses that would have been allowed otherwise, and the policy is enforced at the kernel level. However, there are a number of limitations with these approaches regarding the monitoring of sensitive data leaks. First, access control mechanisms cannot block indirect information flows but instead only control the legality of access to resources. The actual content of resources may illegally flow towards other processes through IPC, files or other objects. Another limitation of access control mechanisms in this context is that they block illegal accesses and thus modify the system's behavior, possibly breaking functionalities. In some situations, users have to modify the security policy in order to be able to perform manual actions, which may lead them to disable the security mechanisms.

3.2.1 Information flow control and taint analysis

In 1973, the Bell-LaPadula model was introduced (LaPadula & Bell 1973), with the primary goal of protecting confidentiality. It is also known as Multilevel Security, and systems that implement it are called Multilevel secure or MLS systems (Anderson et al. 2001). In this model, subjects and objects are labeled with a security level, which represents their sensitivity or clearance. Any information flow from a high security classification to a lower security classification is illegal (Bell & LaPadula 1976, Department of Defense 1987, Foley et al. 2006). Implementations of MLS try to accurately observe data manipulations in order to prevent illegal information flows. Operating systems with MLS implementations include SELinux, FreeBSD, Solaris and BAE XTS-400. In 1976, Denning introduced "a lattice model of secure information flow" (Denning 1976). She defines it as a mathematical framework suitable for formulating the requirements of secure information flow among security classes. Most of the lattice-based information flow models can be represented in Denning's framework. Models of decentralised information flow control based on lattice models (see Section 3.2.1) in operating systems such as Histar (Zeldovich et al. 2006) and Asbestos (Efstathopoulos et al. 2005) provide an alternative to MAC but are still based on access control mechanisms. Flume (Krohn et al. 2007) is a Linux implementation of decentralized information flow control based on Asbestos labels, and contrary to the previous models, it uses standard OS abstractions.

The protection of sensitive information is becoming a serious concern. Recent works regarding the monitoring of private information include Panorama (Yin et al. 2007), TaintCheck (Newsome & Song 2005) and TaintDroid (Enck et al. 2010). Panorama is a system-wide information flow tracking model based on dynamic taint analysis. TaintCheck dynamically taints incoming data from untrusted sources (*e.g.* network) and detects when tainted data is used in any way that could be an attack. Both use full system emulation at the instruction level to provide very fine-grained approaches. However, the main limitation of such instruction-level models is a very high penalty in terms of performances, a slowdown of 20 times in average when using Panorama, and a slowdown of 1.5 to 40 times when using TaintCheck, according to their respective authors. TaintDroid(Enck et al. 2010) is an information flow tracking system for realtime privacy monitoring on smartphones. It is based on taint marking at four different levels of granularity, respectively at the variable, message, method and file levels. TaintDroid has a performance overhead of 14% on CPU. However, TaintDroid is focused on the Android platform using the Dalvick interpreter and therefore it does not apply to native applications, which represent most of the software present on standard desktop operating systems.

3.2.2 Host-based intrusion detection

Where intrusion detection systems (IDSs) are often network related and based on network traffic signatures, there also exist host based IDSs, observing operating system events and raising alerts when suspicious behaviours are observed. Network IDSs are not practical to track sensitive data leaks for the same reason as firewalls and deep packet inspection, as those do not have any knowledge of application level information. This makes it difficult to define signatures that can identify sensitive information.

Policy-based IDSs are anomaly detection IDSs following a "default-deny" approach. A number of previous works exist in this domain, using sandboxing mechanisms at the language level (Inoue & Forrest 2002) or via Kernel based reference monitors such as BlueBox, REMUS, LIDS and Ko et al.'s system wrappers (Chari & Cheng 2003, Habib 2006, Bernaschi et al. 2002, Ko et al. 2000). Similar sandboxing mechanisms also exist in user space, namely system introspection (Wagner 1999, Jain & Sekar 1999).

Contrary to access control mechanisms, such approaches are permissive: they do not block information flows and thus do not modify the system behavior. However, they are inadequate for tracking information flows involving sensitive data. These models of intrusion detection efficiently monitor access to resources when subjects access it but they have a common limitation with access control mechanisms: once access to a resource has been granted, they do not monitor any further information flow towards other processes or system objects.

4 Detection of confidentiality violations

Taint marking techniques along with an information flow policy have been used in previous works for host-based intrusion detection in Blare (Stéphane

Geller, Christophe Hauser, Frédéric Tronel, Valérie Viet Triem Tong 2011). Our current research builds on this system for the detection of confidentiality violations through untrusted applications over the network. This work is based on a subset of the Blare model, that we extended with support for network sockets and a *network policy*. Blare is able to dynamically observe information propagation. Adding support for networking makes it possible to monitor outgoing information while being aware of the involved applications and data.

4.1 Summary of the Blare model

Blare labels information with tags. Objects of the operating system which may contain information (such as files, processes, IPC etc.) are called *containers* each of which has a so-called *information tag*. Every time an information flow occurs towards a *container*, its *information tag* is updated to reflect a maximal estimation of its possible new content[1]. Such tags use meta-data to describe information, and a distinction is made between passive stored data and active code being executed by a process. The set of all passive data in a system is noted $\mathcal{I}$ and the set of all active code is noted $\mathcal{X}$. This distinction is motivated by Denning's assumption that "processes are the active agents responsible for all information flows" (Denning 1987). *Information tags* are sets of meta-information describing the content of containers, namely any combination of $\mathcal{X}$ and $\mathcal{I}$, that we denote as $\wp(\mathcal{I} \cup \mathcal{X})$[2]. Processes are the result of the execution of binary programs, most likely (but not necessarily) stored on disk. Recall the distinction between active and passive information. Passive information stored on disk is labelled with meta-data from $\mathcal{I}$. Every element of $\mathcal{I}$ has an image in $\mathcal{X}$ through a function $exec()$ characterizing the execution. Since we do not have any a-priori means to know if information is executable or not, all information exists in both sets. When a binary containing executable information ($i \in \mathcal{I}$) is executed, the *Information tag* of the new process is initialized to ($x = exec(i)$). This indicates which code is currently being run by the process. After this, its *information tag* is updated after every information flow as described previously.

4.2 Network extension

We have developed an extension of Blare supervising network interactions. Network sockets are information channels, and we track information flowing towards them. There are different families of sockets, including *UNIX domain sockets* and *internet sockets*. The latter are used to communicate with untrusted remote hosts through the internet, and we focus on their usage by userspace applications. Sockets by themselves are not labelled, as we consider those as part of the process memory. Instead, tracking is performed when processes actually send information through those information channels.

4.2.1 Network policy tag

The policy for communicating with internet sockets is defined globally through a unique shared *network policy tag*. The *network policy tag* is a tuple defining which combinations of information may legally leave the local system through internet sockets, and optionally which applications may communicate, as well as which information each application may communicate (per-application profiles).

A *network policy tag* is defined as follows:

$$P_{net} \in \wp(\wp(\mathcal{I} \cup \mathcal{X}))$$

It is a tuple of sets that can contain any combination of elements from $\mathcal{I}$ (passive data) and $\mathcal{X}$ (running code).

The following properties apply to P_{net}:

- Elements of $\mathcal{I}$ in the sets of P_{net} represent mutually exclusive sets of data which can legally flow out of the system (i.e., only one of the sets is legal at one time).
- Elements of $\mathcal{X}$ in the sets of P_{net} represent *supervised*[3] code which is allowed to communicate through internet sockets.
- Any combination $A \subseteq \wp(\mathcal{I} \cup \mathcal{X})$ in the sets of P_{net} defines a profile for applications, where elements of $\mathcal{I}$ define which data can be sent over the network, and elements of $\mathcal{X}$ define which running code may send that information.

4.2.2 Legality of network information flows

When a process sends information through a socket, a legality verification is performed on its current *information tag* against the global *network policy tag*. The information flow is legal if and only if the content of its *information tag* is contained in one of the subsets of the *network policy tag*.

Definition 1. *For any information tag containing a set of data $S \subseteq \wp(\mathcal{I} \cup \mathcal{X})$, the boolean relation $Legal_{net}$ is defined as follows:*

$$Legal_{net}(S) \Leftrightarrow \exists p \in P_{net} | S \subseteq p$$

4.3 Practical use cases

Our approach covers the following use cases. In the following, the term *labelling* refers to the action of attaching a unique *information tag* to a file.

4.3.1 All sensitive data must stay local

In this use case, the user of the system wants all of the sensitive data to stay local. Any network transfer of those data is a violation of the policy and our extended version of Blare will report a privacy violation alert. This can be accomplished by only labelling sensitive data (files) that should never flow out of the system. By defining an empty *network policy tag*, no data can legally flow out through network sockets, and the user will be notified every time a socket sends such tainted data over the network.

$$P_{net} = \{\{\}\}$$

[1] The new *information tag* is the union of the source's *information tag* and the destination's *information tag*: a conservative estimate that is safe but may be an overestimation, reflecting a 'worst case' scenario where the complete content of the source is copied to the destination. This is necessary because it is impractical to observe actual information flows (Zeldovich et al. n.d.).

[2] Powerset $\wp(A)$ denotes all the subsets of A.

[3] The corresponding binary file is labelled with an *information tag*.

4.3.2 Sensitive data may be sent over the network only through trusted applications

In this use case, the system contains both trusted and untrusted applications, as well as some sensitive data which may flow over the network only through trusted applications. This can be accomplished by labelling all the binary applications on the system along with all the sensitive data. The *network policy tag* is set to match the union of all the *information tags* of the binaries and those of sensitive files on the filesystem. In this case, the *network policy tag* is a tuple with only one set.

$$P_{net} = \bigcup_{i=1}^{N} (S \cup C)$$

Here S is the set of all the sensitive data and C the set of all trusted code.

4.3.3 Per-application profiles

In this use case, the system contains both trusted and untrusted applications, and each trusted application may send a different set of sensitive data over the network. This can be accomplished by labelling all the binary applications on the system along with all the sensitive data. Then, the *network policy tag* is a tuple of several sets such as:

$$P_{net} = \{\bigcup_{i=1}^{N} (s \cup c) | s \subseteq S, c \subseteq C, legal(c, s)\}$$

where $legal(a, b)$ states that the application a is allowed to send information b over the network.

4.4 Dynamic policy changes

Taint marking can sometimes lead to a growing number of false positives due to the fact that tainted data remains tainted until the system reboots, and information flows keep propagating tainted data between objects of the operating system. This may lead to repetitive alerts about the same data leaking. Furthermore, the user or administrator may decide to declassify some information that he or she previously considered as private, and allow it to flow over the network.

For this reason, users can decide to modify the policy on the fly while the system is running. New sets can be dynamically added to the *network policy tag* at runtime. Several situations may occur:

- Only sensitive data has been labelled, and may not flow over the network. There are no trusted applications. In this case, the user can permanently neutralize alerts concerning a set of sensitive data S by adding a new tuple S to the *network policy tag*.
- Both sensitive data and trusted application's code have been labelled, and the user wants to neutralize alerts concerning one set of sensitive data S leaked by processes running code C. This can be performed by adding a new set to the *network policy tag* containing $(C \cup S)$.

4.5 Implementation

The reference monitor for this current model has been implemented in Linux version 2.6.39, as a Linux Security Module (LSM). This new implementation builds on a model described earlier in (Stéphane Geller, Christophe Hauser, Frédéric Tronel, Valérie Viet Triem Tong 2011) along with our network extension presented in this paper, and has been written from scratch using the C programming language. *Information tags* are implemented as linked lists of 64-bit signed integers, where positive values represent the set of passive data $\mathcal{I}$ and negative values represent the set of active data (*i.e.,* code of running processes) $\mathcal{X}$. The *Network policy tag* is implemented as a linked list of legal sets, where each set is a red black tree for fast $o(log(n))$ lookups. Labels are written in the security namespace of the extended attributes of the filesystem. A userspace interface is exported through securityfs[4] to load the *network policy tag* in kernelspace. Supervised socket families are `AF_INET` and `AF_INET6`. A userspace daemon reports alerts to the user via the libnotify library. Userspace tools allow us to manipulate filesystem extended attributes to set and edit *information tags*.

5 Experiments

5.1 Data leaks through a web browser

The following scenario shows how our new model and implementation can detect confidentiality violations by untrusted code interpreted by a Web browser. Web browsers were initially simple applications displaying HTML content to the final user, but those have evolved into complex applications running JavaScript and other interpreted languages on the client machine, inevitably exposing user data to a number of real threats. In this scenario, a client is running a modified Linux kernel with our implementation of Blare with the presented network extension. The client visits a malicious web page using Mozilla Firefox 3.5 and the Java runtime environment plugin (JRE) version 6 update 10. This version is subject to the "Java calendar deserialization" vulnerability (CVE 2008-5353) that may lead to the execution of arbitrary code by an attacker. The client executes malicious Java code exploiting this issue and embedding a payload that allows the attacker to get a remote shell on the machine.

Assume the folder`/home/alice/confidential/` contains 64 confidential files. We labeled these files as being confidential, and assigned an *information tag* containing a unique identifier between 1 and 64 to each of them. The *information tag* of these files is a set containing one unique identifier, *e.g.,* {1}. This experiment is similar to the use case "all sensitive data must stay local" introduced in Section 4.3.1. We defined an empty *network policy tag* as follows :

$$P_{net} = \{\{\}\}.$$

In this configuration, any application sending any of the labelled files to any remote host is a security policy violation and triggers an alert. Now we visited a crafted web page *http://www.malicious-host/malicious-page.html* embedding a malicious Java applet containing an attack against the previously mentioned vulnerability. This malicious page causes Mozilla Firefox to execute the Java virtual machine (JVM) in a separated process, which in

[4] Securityfs is based on sysfs and is used by the LSM modules, generally mounted as /sys/kernel/security.

turn interprets the Java code containing a remote shell allowing the attacker to connect to the local machine. As the attacker accesses labelled files of the local filesystem, the *information tag* of the process running Java is updated with *information tags* of the files it reads. At the moment when it sends information through a socket, our kernel reference monitor considers that the data being sent contains information from the files it previously read, and proceeds to a lookup throughout the *network policy tag* to ensure this behavior is allowed by the user. For every illegal attempt to illegally send information by the Java process, we were warned by the reference monitor with the following message:

```
[BLARE_POLICY_VIOLATION] Illegal information
sent to socket by
process [PID] running java
```

5.2 Evaluation of performances

The following is an evaluation of our implementation in terms of performances. We uncompressed a Linux kernel source tree and used it as a dataset containing 39048 files, that we individually labeled with a unique *information tag*. The machine we used is a Pentium 4 3.0 Ghz with 2.5 Gb of RAM. We evaluated the performances of our kernel following the scenario "all sensitive data must stay local" as presented in Section 4.2.

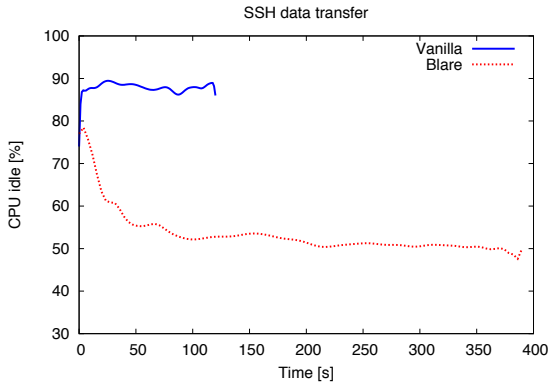

Figure 2: CPU overhead on SSH transfer

Figure 2 compares the CPU idle time when using Linus Torvald's kernel (that we call Vanilla) and the Blare kernel. As expected, the Vanilla kernel gives lower CPU overhead during the transfer (higher CPU idle value). Our security framework adds 30% to 40% of extra overhead to the data transfer.

Figure 3 compares the memory overhead of our kernel and makes a comparison with a Vanilla kernel executing the same file transfer operation. As Blare is attaching meta-information to every system object, the memory consumption remains higher by 30% on average when using our Kernel.

5.2.1 Overall completion time

The overall completion time was 300% longer with our kernel than with the Vanilla kernel. This limitation is due to a bottleneck at the filesystem level in our prototype. Extended attributes of the filesystem are used extensively in our implementation with no optimization. We believe that the overall performances of our system can be improved by optimizing the current prototype.

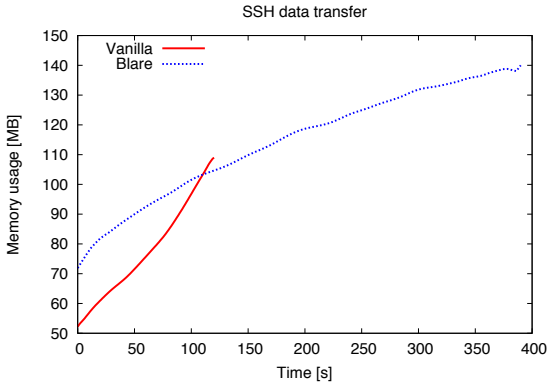

Figure 3: Memory overhead on SSH transfer

5.2.2 Detection rate

When experimenting with an empty *network policy tag*, Blare reports all labelled information that is leaving the operating system with no *false negatives*. By design, our conservative approach, as described in Section 4.1 does not allow *false negatives*. However, a variable number of *false positives* may occur, depending on the presence of indirect flows where Blare over estimates the actual content. Due to the impractical aspects of such an evaluation, we have not performed any comprehensive study of the false positive rate in this study. This aspect will be further evaluated in future work.

6 Discussion

6.1 Advantages

By design, our model does not involve *false negatives* as our tainting technique makes an overestimate of any possible content residing in system objects after any information flow occurs. Furthermore, we do not only monitor network traffic, but any information flow between objects of the operating system.

6.2 Usability

We consider the presence of *false positives* in this model as an acceptable fact in most situations given that it is meant to protect from situations which should happen by no means, based on the principle of non-interference (Ko & Redmond 2002). The false positives rate can be improved by filtering alerts in userspace, for instance any sequence of false positives triggered by the same event can safely be discarded after the event has been reported. A more comprehensive evaluation of the false positive rates will be studied in future work.

This model does not replace access control mechanisms, nor enforce any security policy but instead helps to ensure no unwanted behaviour happens between some defined sets of data and the network. The situation where a web-browser accesses some personal information is a good example of our goals: where access control could have been used to block this particular access in the first place, it does not prevent an application from indirectly accessing the same information by another channel (shared memory, IPC with another application etc.). Furthermore, in this example we focus on the fact that this information should not leave the system through the network, therefore

no alert would be raised for an application that accesses the information but does not send it over.

7 Conclusion

Most of today's computer software includes a number of third party applications and plugins from untrusted sources. Users have no control over the actions such software can perform, and no guarantee regarding the confidentiality of their data. This article presented an approach of confidentiality violation detection based on dynamic data tainting to address this issue. We implemented a reference monitor in the Linux kernel allowing for system-wide information flow tracking. We presented a practical example showing how it is possible to detect confidentiality violations using our model.

References

AbuHmed, T., Mohaisen, A. & Nyang, D. (2008), 'A survey on deep packet inspection for intrusion detection systems', *Arxiv preprint arXiv:0803.0037* .

Anderson, R. J., Stajano, F. & Lee, J.-H. (2001), 'Security policies', *Advances in Computers* **55**, 186–237.

Bell, D. E. & LaPadula, L. J. (1976), Secure computer system: Unified exposition and multics interpretation, Mtr-2997 (esd-tr-75-306), MITRE Corp.

Bernaschi, M., Gabrielli, E. & Mancini, L. V. (2002), 'Remus: a security-enhanced operating system', *ACM Trans. Inf. Syst. Secur.* **5**, 36–61.

Chari, S. N. & Cheng, P.-C. (2003), 'Bluebox: A policy-driven, host-based intrusion detection system', *ACM Trans. Inf. Syst. Secur.* **6**, 173–200.

Denning, D. E. (1976), 'A lattice model of secure information flow', *Commun. ACM* **19**(5), 236–243.

Denning, D. E. (1987), 'An Intrusion-Detection Model', *IEEE transaction on Software Engineering* **13**(2), 222–232.

Department of Defense (1987), 'Trusted network interpretation of the DoD TCSEC (red book)', NCSC-TG-005.

Efstathopoulos, P., Krohn, M., VanDeBogart, S., Frey, C., Ziegler, D., Kohler, E., Mazières, D., Kaashoek, F. & Morris, R. (2005), Labels and event processes in the asbestos operating system, *in* 'SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles', ACM, New York, NY, USA, pp. 17–30.

Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P. & Sheth, A. N. (2010), Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones, *in* 'Proceedings of the 9th USENIX conference on Operating systems design and implementation', OSDI'10, pp. 1–6.

Foley, S. N., Bistarelli, S., O'Sullivan, B., Herbert, J. & Swart, G. (2006), Multilevel security and the quality of protection, *in* 'Proceedings of First Workshop on Quality of Protection', Springer, p. 2006.

Georges, L., Tong, V. V. T. & Mé, L. (2009), Blare tools: A policy-based intrusion detection system automatically set by the security policy, *in* 'Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection (RAID 2009)'.

Habib, I. (2006), 'Getting started with the linux intrusion detection system', *Linux J.* **2006**.

Harada, T., Horie, T. & Tanaka, K. (2003), 'Access policy generation system based on process execution history', *Network Security Forum* .

Ingham, K. & Forrest, S. (2002), A History and Survey of Network Firewalls, Technical report.
URL: *http://www.cs.unm.edu/~treport/tr/02-12/firewall.pdf*

Inoue, H. & Forrest, S. (2002), Anomaly intrusion detection in dynamic execution environments, *in* 'Proceedings of the 2002 workshop on New security paradigms', NSPW '02, ACM, New York, NY, USA, pp. 52–60.

Jain, K. & Sekar, R. (1999), User-level infrastructure for system call interposition: A platform for intrusion detection and confinement, *in* 'Network and Distributed Systems Security Symposium'.

Ko, C., Fraser, T., Badger, L. & Kilpatrick, D. (2000), Detecting and countering system intrusions using software wrappers, *in* 'Proceedings of 9th USENIX Security Symposium (SEC 2000)'.

Ko, C. & Redmond, T. (2002), Noninterference and intrusion detection, *in* 'Proceedings of the 2002 IEEE Symposium on Security and Privacy', pp. 177–187.

Krohn, M., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M. F., Kohler, E. & Morris, R. (2007), Information flow control for standard os abstractions, *in* 'Proceedings of the 21st Symposium on Operating Systems Principles', Stevenson, WA.

LaPadula, L. J. & Bell, D. E. (1973), Secure computer systems: A mathematical model, MTR-2547 (ESD-TR-73-278-II) Vol. 2, MITRE Corp., Bedford.

Newsome, J. & Song, D. (2005), Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software, *in* 'Proceedings of the Network and Distributed System Security Symposium (NDSS 2005)'.

Novell/SUSE (n.d.), Apparmor, application security for linux, Technical report.
URL: *http://wiki.apparmor.net*

Schaufler, C. (n.d.), The simplified mandatory access control kernel, Technical report.
URL: *http://schaufler-ca.com*

Stephen Smalley, C. V. (2002), Implementing SELinux as a Linux Security Module, Technical report, NAI Labs.

Stéphane Geller, Christophe Hauser, Frédéric Tronel, Valérie Viet Triem Tong (2011), 'Information flow control for intrusion detection derived from mac policy', *Proceedings of the IEEE International Conference on Computer Communications (ICC)* .

Wagner, D. A. (1999), Janus: an approach for confinement of untrusted applications, Technical report, Berkeley, CA, USA.

Watson, R. & Vance, C. (2003), The TrustedBSD mac framework: Extensible kernel access control for freebsd 5.0, *in* 'In USENIX Annual Technical Conference', pp. 285–296.

Wright, C., Cowan, C., Smalley, S., Morris, J. & Kroah-Hartman, G. (2002), Linux security modules: General security support for the linux kernel, *in* 'USENIX Security Symposium', pp. 17–31.

Yin, H., Song, D., Egele, M., Kruegel, C. & Kirda, E. (2007), Panorama: capturing system-wide information flow for malware detection and analysis, *in* 'Proceedings of the 14th ACM conference on Computer and communications security', CCS '07, pp. 116–127.

Zeldovich, N., Boyd-Wickizer, S., Kohler, E. & Mazières, D. (2006), Making information flow explicit in histar, *in* 'OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation', USENIX Association, Berkeley, CA, USA, pp. 263–278.

Zeldovich, N., Kannan, H., Dalton, M. & Kozyrakis, C. (n.d.), Hardware enforcement of application security policies using tagged memory., *in* R. Draves & R. van Renesse, eds, 'Proceedings of OSDI', USENIX Association, pp. 225–240.