Preface

TBD

Sipoo, January 11, 2018,

Elena Reshetova

Contents

List of Publications

This thesis consists of an overview and of the following publications which are referred to in the text by their Roman numerals.

I Kostiainen, Kari and Reshetova, Elena and Ekberg, Jan-Erik and Asokan, N.. Old, new, borrowed, blue: a perspective on the evolution of mobile platform security architectures. In *Proceedings of the First ACM Conference on Data and Application Security and Privacy*, Location, pages, and other detailed information, Month 2011.

II Reshetova, Elena and Bonazzi, Filippo and Nyman, Thomas and Borgaonkar, Ravishankar and Asokan, N. Characterizing SEAndroid Policies in the Wild. In *2nd International Conference on Information Systems Security and Privacy*, Location, pages, and other detailed information, October 2016.

III Reshetova, Elena and Bonazzi, Filippo and and Asokan, N. SELint: an SEAndroid policy analysis tool. In *ICISSP*, Location, pages, and other detailed information, October 2017.

IV Reshetova, Elena and Karhunen, Janne and Nyman, Thomas and Asokan, N. Security of OS-Level Virtualization Technologies. In *Nordic Conference on Secure IT Systems*, Location, pages, and other detailed information, October 2014.

V Reshetova, Elena and Bonazzi, Filippo and Asokan, N. Randomization Can't Stop BPF JIT Spray. In *Network and System Security: 11th International Conference*, Location, pages, and other detailed information, August 2017.

VI Reshetova, Elena and Liljestrand, Hans and Paverd, Andrew and Asokan, N. Towards Linux Kernel Memory Safety. Submitted to *ACM Conference on Data and Application Security and Privacy*, March 2018.

Author's Contribution

Publication I: "Old, new, borrowed, blue: a perspective on the evolution of mobile platform security architectures"

The author did this and that

Publication II: "Characterizing SEAndroid Policies in the Wild"

The author did also this and that

Publication III: "SELint: an SEAndroid policy analysis tool"

The author did also this and that

Publication IV: "Security of OS-Level Virtualization Technologies"

The author did also this and that

Publication V: "Randomization Can't Stop BPF JIT Spray"

The author did also this and that

Publication VI: "Towards Linux Kernel Memory Safety"

The author did also this and that

1. Introduction

Find a nice quote here.

1.1 Motivation

Nowadays platform security is an essential component of many devices around us: from smartphones and tablets to smart watches and glasses, from home entertainment and video surveillance systems to smart home appliances, from in-vehicle infotainment to flying drones. More and more functionality that used to be carried our manually by humans is automated with interconnected devices. This trend brings many additional challenges into standard platform security architectures that originally have been developed for solid and relatively closed systems, such as servers, military devices etc. Mobile devices, such as smartphones, were the first new wave of devices that required a totally different approach for designing platform security due to their open nature, i.e. ability for a user to install third-party applications, as well as a wide range of users that it was targeting. A designer of platform security solution could not anymore assume that a user of a device is an experienced system administrator understanding a wide range of security risks and enforcing mechanisms. A mobile device user not only lacks this knowledge, but also in many case explicitly not willing to be involved in any of such matters or decisions: he or she just wants to accomplish a simple task of sending a message or installing a popular game. The situation gets even more severe as we start looking into various in-vehicle infotainment systems where user-driven open entertainment system might be partly co-allocated with car's instrument cluster devices with a very strict safety requirements and regulations.

There is no such thing as the perfect platform security architecture, its design and implementation varies greatly based on targeted environment, use cases, device capabilities, cost factors and prioritized set of risks and threats. A lot of prior research both in academia and industry examines this problem from all different angles and aspects, but the practice shows that achieving a good

balance is still hard. Many attacks are published regularly that find holes in or abuse different platform security mechanisms and lead to severe implications on user's privacy, overall safety, denial of service and others. The producers of various devices, OEMs, in turn are in a very hard position of delivering their solutions fast (due to a very competitive pace of modern industry) and at the same time satisfying security requirements of various involved stakeholders from ordinary users to various legislation.

1.2 Objectives

The goal of this thesis is to look at different practical platform security challenges that modern mobile and embedded systems are facing today and will face in the nearest future and try to develop mechanisms and techniques that can help OEMs deliver more secure devices. While doing this we want to make sure that whenever a certain mechanism or solution is proposed, it satisfies the following requirements:

1. **Security**. The proposed mechanism must provide sufficient security level for the given set of use cases. There is no aim of reaching a perfect security if it makes the solution too unpractical for deployment or fails to satisfy the below non-security-related requirements.

2. **Practical deployability**. A mechanism or technique that is hard to deploy in practice is of little use in the industry. Instead the end goal of each proposed measure should be its integration and practical use in the respective area of interest. While it is not always possible to achieve this goal, it must be shown that the proposed solution satisfies all the standard deployability requirements of the field, such as performance, development life cycle, scalability, etc.

3. **Usability**. The solution must be easy to use and understand for an ordinary user of the system. Latter might not necessary mean the end user of a device, but can also be an application developer, an access control policy writer, a kernel subsystem maintainer etc. These users usually have very different knowledge base and backgrounds, but in order for a solution to be taken into use and used correctly, it is essential to meet the usability requirements of all its intended users.

1.3 Outline

This dissertation is based on the six original publications which are grouped into three main areas, all around various aspects of platform security. Each chapter

starts with the relevant background and ends with a short discussion section that evaluates the contribution as well as lists open problems in the area.

Chapter 2 presents an overall view on the main aspects of any platform security solution and includes Publication I that is a survey of the platform security architectures for the most common mobile operating systems. While two out of four considered OSes, Symbian and MeeGo, have been terminated since, they played a big role in the development of mobile platform security architectures and methods. Chapter 2 ends with a motivation for selecting the two specific focus areas for this dissertation, described in the next two chapters.

Chapter 3 takes a deep look into the process isolation and access control methods employed by modern operating systems and encompasses Publications II, III and IV. Publication II examines the relatively new and raising method for process isolation that is based on the OS-level virtualization and shows its shortcomings against the traditional mandatory access control schemes. Publication III takes a deep look on the mandatory access control mechanism (SEAndroid), employed by the most popular mobile operating system at the moment when OEMs had to go through the challenge of learning to create access control policies. The publication shows the difficulties that developers were facing when trying to create the correct policy for their components, as well as most common pitfalls. It also presents a list of practical tools that can help OEMs, developers and security researchers to create better access control policies. Finally Publication IV presents one of such tools, SELint, that was considered the most helpful for this task.

Chapter 4 focuses on a very different challenge: a need to be able to have the core of an operating system, OS kernel, secured against the various attacks. This is a very important area, because an OS kernel is becoming more and more attractive target for the attackers due to both its high privileges and recent tightening of the userspace security. Publication V shows a powerful attack against a very commonly used Linux kernel subsystem, Just In Time (JIT) Compiler for Berkeley Packet Filter, which served as a motivator for a stricter hardening of this subsystem. In turn Publication VI proposes two methods for addressing the Linux Kernel OS memory safety and therefore preventing common Linux OS kernel vulnerabilities, such as various buffer overflows and use-after-frees.

Finally, Chapter 5 presents an overall evaluation of the methods proposed in this dissertation against the objectives set in Section 1.2 and attempts to give an outlook on the whole future mobile and platform security challenges.

2. Mobile and Embedded Platform Security

Here I can give an overview of a typical design that a mobile/embedded platform security follows.

Here publication I is refereed and explained. Here motivate that access control/process isolation, as well as OS kernel hardening are one of the biggest practical challenges for today platform security architectures despite millions of past works both in academia and industry. The section would end in stating two main areas that are of the focus of this thesis: access control and isolation and kernel OS hardening.

3. Process isolation and access control policies

Isolation is easy, sharing is hard.
find original source or use Casey's name

The notion of access control (AC) and application/process isolation is one of the oldest in the history of the information security with first pioneering works being done back in 80-ies. Initially the main goal of AC methods was to isolate different system's users with different level of access to the system assets, but as computers and other IT devices started to become more and more single-user purpose, the focus has shifted towards isolation of system processes and applications. This trend was especially noticeable in various mobile platform security architectures, when users got an ability to install third-party software on their mobile devices because it required a stricter isolation between system's core platform and services (trusted set of entities) and user installable services and applications (untrusted and potentially malicious).

A typical AC system can be viewed as a collection of one or more AC reference monitors that are consulted for AC decisions whenever any system access (such as operation on a filesystem object) is performed. The AC decision is done based on the AC policy for that AC reference monitor and defines if the access should be allowed or not. The AC policy is written in a form of AC rules that are defined in terms of *subjects* (active entity performing the access), *objects* (passive entity that is being accessed) and *access types* (what type of access is being attempted, such as read, write etc.). Subject and objects are also typically grouped into a set of *access control domains* using some identification or labeling scheme, allowing policy writers to arrange related and interconnected OS components into logical blocks. Typically most of the AC systems employ the principle that by default any access is denied unless explicitly allowed by AC policy rules or both subject and object are belonging to the same access control domain.

Over the decades the topic of process and application isolation was a focus for tremendous amount of academia and industrial research works with many systems and mechanisms proposed to address the problems arising when designing and implementing AC systems. On a high level these problems can be arranged into the following categories that are all interconnected and affecting each other:

- **Granularity.** One can design and deploy AC mechanisms with very different granularity levels following the principle of least privilege. On one side is placing each individual process or thread into its own access control domain, and on the other side only creation of high-level access control domains that would isolate the trusted part of the system from user-installable and less trusted applications. It is clear that a fine-grained AC policy allows creating a more precise set of AC rules for a given AC object, however it usually directly negatively impacts other factors like manageability and performance making policy writers and designers to search for some intermediate balance.

- **Manageability.** Most modern mobile and embedded OSes change at a high pace: new system services and APIs being introduced with each release, existing services might be fully redesigned, new use cases might be introduced etc. Same applies for the system's AC policy: it must adapt and change with the system to make sure it not only correctly reflects the current state of the system, but also correctly isolates and protects its entities. In addition one needs to take into account how OS is being developed. For example if OS changes are done in distributed fashion by different independent set of development teams, the policy needs to be modular enough to allow changes to be performed relatively independently and in parallel. One just cannot assume that there is a single person that can manage all policy changes centrally.

- **Performance.** Typically an access control system is integral part of an OS and for every system access, an access control decision needs to be made based on available parameters. This makes an AC mechanism a very run-time performance sensitive component and care needs to be taken to make the overhead acceptable for system use.

Part of this thesis investigates how the modern AC mechanisms attempt to find a balance in the above categories, what are the concrete problems that they are facing in a struggle to find this balance, as well as to understand what tools or techniques can be helpful in practice to solve these problems. For this purpose the thesis selects one of the most popular mobile operating system, Android and its mandatory AC mechanism, SEAndroid for a detailed evaluation as the most used mandatory AC mechanism on mobile and embedded Linux-based OSes. Another focus area of this thesis is an attempt to look into alternative ways how process and application isolation can be done using various OS-level virtualization techniques (aka application or system containers) that during couple of last years became a fashionable alternative to traditional AC mechanisms. The main research question that this thesis attempts to answer with regards to the OS-level virtualization is whenever this technique can provide a similar level of security that traditional mandatory AC mechanisms.

3.1 Mandatory access control on modern systems: SEAndroid

Over the past decades, Android has become one of the most common OSes for mobile devices. As the amount of devices running Android has been increasing, so was the amount of malware targeting this operating system. Back in 2011 it became clear that the discretionary Android permission access control model is not able to provide the required level of security and the work started on adapting the well-known desktop mandatory access control mechanism, SELinux, for the purpose of Android OS. The resulting mechanism, SEAndroid, was largely based on SELinux, but contained some adjustments and additions required for the supporting some of Android-specific mechanisms, such as Binder Inter Process Communication (IPC). The biggest change was the SEAndroid reference policy, which was written from scratch and due to significant differences in the Android and desktop userspace layer, as well as a desire to have a small and compact policy. The initial policy and enforcement points were added to the Android 4.3 release back in 2012, but it wasn't until Android 5.0 Lollipop release when SEAndroid was required to be enabled on all devices in the system-wide enforcing mode, which forced OEMs to start working on SEAndroid policies.

Traditionally the Android ecosystem works in a way that Google delivers the reference code for the Android OS in a form of open source AOSP project, which is taken by different Android OEMs and customized at various levels starting from HW adaption, new OEM-specific drivers and system services, and ending up some user facing applications. The reference SEAndroid policy, provided by the AOSP project, does not obviously cover these additions and customizations, and therefore OEMs need to adjust the provided SEAndroid policy not only to get to a desired level of mandatory access control enforcement, but also merely to make their customized Android OS work correctly.

The publication II of this thesis starts by examining 8 different SEAndroid policies from various Android OEMs that was created for Android 5.0 release and provides extensive characteristics of these policies in terms of OEM-performed changes, policy size and complexity and etc. Table 1 from publication II shows that all OEMs extensively modify the reference policy resulting in overall policy size growth, as well as all other policy attributes and access control rules. Next the publication II manually analyzes the characteristics of each of these 8 policies and builds a set of typical pitfalls that all OEMs seems to make when performing their policy additions. These include overuse of default types, where many OEMs leave SEAndroid default labels[1] for their newly added system objects, which typically leads to aggregation of big amount of non-related objects under a single default label and consequently leads to the violation of the principle of least privilege since many different domains might need to have legitimate access to the objects under default labels. Another typical pitfall pattern is aggregation of OEM-created applications under the reference policy predefined

[1] if an object is not given an explicit label by the SEAndroid policy, it is assigned one of the default labels, such as `unlabeled`, `device`, `default_prop` etc.

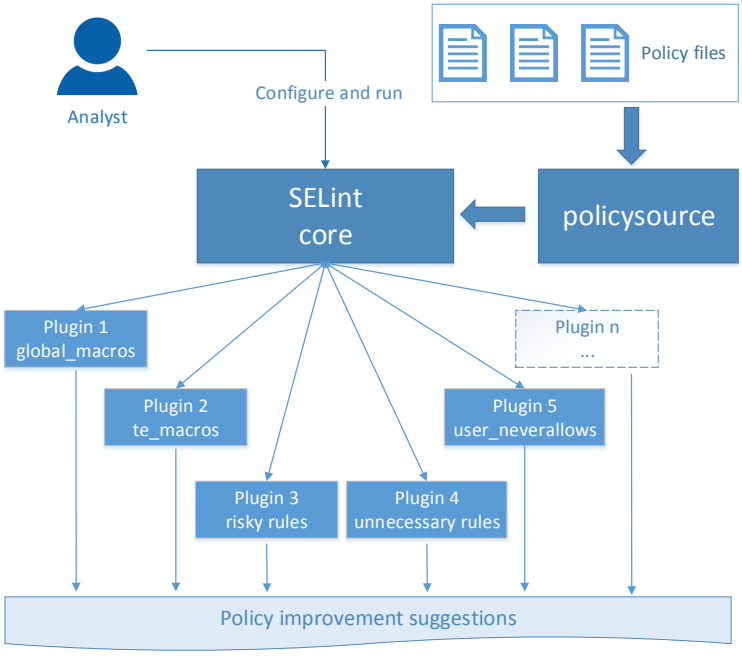


Figure 3.1. The architecture of SELint (From Publication III)

application domains, such as `platform_app` or `system_app` instead of creation of smaller, OEM-specific application domains. This leads to the enormous grow of such default domains (for example one OEM policy had 900 allow rules associated with `system_app` domain vs. 46 in the AOSP reference policy) and inability to prevent privilege escalation between different OEM applications placed in these domains. Other OEMs pitfalls include seemingly useless or forgotten rules, as well as rules that are dangerous for the overall security of the system (such as granting access to rather sensitive parts of the system to the untrusted domains), which indicate that OEMs struggled to deliver the comprehensive and secure SEAndroid policies on their devices.

Remember to either get the copyright permission or modify the figure

As a result of above study the publication II proposes a set of tools that should help both OEMs and security researchers to develop better SEAndroid policies without the above-mentioned pitfalls, including various policy analyzing and virtualization tools. It also implements one of such tools, a live SEAndroid policy analyzer (SEAL), that allows performing different policy queries not only based on the actual policy loaded on the devices, but also also a run-time device state, i.e. running processes and services, filesystem labeling etc. SEAL allows to get a fast answer on various questions that a SEAndroid policy writer or analyzer might have, such as "What files can a given running process access on a device?" or "What processes can access a given object?". This significantly simplifies the debugging, troubleshooting and studying a given SEAndroid policy and gives a fast way to discover the inconsistencies.

The publication III presents the design and implementation of a more comprehensive SEAndroid tool, SELint, that is designed for an overall improvement of SEAndroid policy development. It operates on SEAndroid policy sources and allows smooth integration into the OEM policy development workflow. The architecture of SELint is shown in Figure 3.1. It is composed from a SELint core, responsible for the heavy processing of input SEAndroid policy files, and an initial set of SELint plugins that operate on a policy representation provided by the core and able to do the policy analysis. The set of plugins is designed to be extensible and adapt to the needs of different OEMs or security researchers. The initial set of plugins includes 5 distinct ones: two plugins (`global_macros` and `te_macros`) verifying correctness of usage different types of SEAndroid policy macros, a `user_neverallows` plugin that allows a OEM-specific verification of additional neverallow rules, a `unnecessary_rules` plugin that attempts to detect various ineffective rules[2] or rules used for debug purposes, and finally a `risky_rules` plugin that can be used to categorize SEAndroid access control domains into set of related ones (untrusted, security-sensitive, core tcb etc.) and display various risk and trust relationships between them in a given policy. For example, it is possible to highlight the access control rules where access to a security-sensitive object is given to a subject belonging to an untrusted domain.

3.2 Application and process isolation using OS-level virtualization

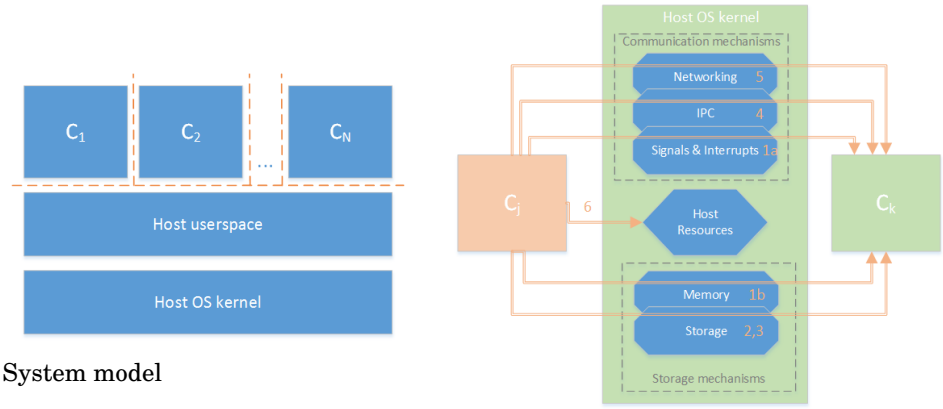
While traditional access control mechanisms and systems aiming for application and process isolation has existed for decades, recent years have seen a raise of an alternative approach, OS-level virtualization techniques, more commonly known as "‘containers"’. While originally developed purely to support various virtualization-based use cases, such as server consolidation or application and resource state management, they later on turned into various security-driven use cases, such as application isolation and Bring Your Own Device (BYOD) scenario[3].

In contrary to the traditional virtualization solutions, like well-known Xen hypervisor or Linux Kernel Virtual Machine (KVM), an OS-level virtualization virtualizes the OS kernel resources available to userspace, as opposite to the actual physical HW resources, and therefore allows processes to share the same host OS kernel. This greatly reduces the performance overhead incurred by the OS-level virtualization and consequently it became possible to use this technology for an efficient isolation of stand-alone applications or their sets.

Parallel to their popularity, there is a constant debate about the security level

[2] some rules in SEAndroid policy are only effective in combination with others, such as domain transition rules when a process wants to transition its domain to a different one upon opening a certain type of object

[3] a single physical device is used for both business and personal purposes that requires a strict isolation between these two environments and limited sharing



(a) System model

(b) Attacker model

Figure 3.2. OS-level virtualization

that such virtualization provides in practice, as well as an attempt to find the gaps in the current implementations. The publication IV of this thesis identifies the security requirements and generic model for a typical OS-level virtualization setup, and then compares a selection of available (both present and historical) solutions with regards to this model.

The system and attackers model for OS-level virtualization are shown in Figure 3.2. The set of containers C1..CN are running on top of shared host OS kernel and userspace layers. The latter one can be either minimal layer only responsible for doing the setup of containers or a full fledged host OS userspace layer. The attacker model assumes that an attacker has a full control over a certain subset of containers and attempts to attack either another subset of legitimate containers running on the same host or host OS itself with either of these three goals: privilege escalation, legitimate container or host compromise or denial-of-service. Each of this can be achieved by one of the attack groups that can be roughly classified based on typical interfaces available to a UNIX-compliant OS (1-6 in Figure 3.2b). These attack groups are then converted into security requirements that each OS-level virtualization solution must fulfill in order to provide strong isolation: separation of processes, filesystem isolation, device isolation, IPC isolation, network isolation and resource management.

The publication IV analyzes 6 OS-level virtualization solutions against the above security requirements and outlines the differences in provided level of isolation. It also summaries the state of OS-level virtualization in the mainline Linux kernel, together with the identified gaps that were limiting the usage of this technology in the environments with strict security isolation requirements. However in the years after the publication IV was done many of these gaps has been either addressed by the mainline Linux kernel or have progressed towards a more robust solution:

- Security namespaces.

- Device namespaces.
- (Pseudo)random number generator devices.
- Hotplug support.
- Cgroups support.

In addition a lot of work has been put into namespacing the rest of Linux kernel security subsystems, such as Integrity Measurement Architecture (IMA), Kernel Keyring etc.

3.3 Discussion

While the implementation of the OS-kernel virtualization in the mainline Linux kernel continues to evolve and improve, the classical mandatory access control schemes are still required to be jointly used in order to provide the highest possible level of isolation and minimize the security risks. Security architects must understand in details the limitations of OS-kernel virtualization in order to make well-grounded choices on a set of isolation mechanisms that their system is required in order to fulfill the security requirements for its use cases. The publication IV of this thesis presents a good model for such evaluation and comparison of available mechanisms, as well as guidance on identifying the missing isolation gaps.

The SEAndroid MAC continues to be the main AC enforcement point of Android OS with all major OEMs now accustomed to the task of working with SEAndroid policies. The evaluation of initial SEAndroid policies done in publication II has been crossed referenced in the official Google SEAndroid documentation as a valuable insight into practical state of things. The SEAndroid tools proposed in publications II and III are being used by the Android community with numerous private forks done on the project code trees and bug reports filled by the users once in a while[4]. The SELint tool fulfills the set of main functional R1 - R4 requirements defined in Section IV of Publication III by exhibiting an acceptable performance overhead, allowing its usage by ordinary users (after an initial expert configuration), being flexible and easy extensible. The detailed evaluation can be found in Section V of Publication III.

[4] https://github.com/seandroid-analytics

4. OS Kernel Hardening

It is easy to see that a central piece of any platform security architecture is the security of the operating system (OS) kernel: any breach in this area almost always leads to the compromise of the whole system, especially if no security hardware support is present. This makes the OS kernel a very attractive target and many recent studies show that adversaries are more and more focusing their efforts on kernel-level attacks [38].

put more references

This is especially true given that many popular mobile and embedded OSes spent considerable effort in tightening the security of their userspace applications and processes [38].

put more references

For example, it used to be pretty common that many userspace daemons run with superuser privileges all the time making it very easy for attackers to obtain these privileges right after compromising a single userspace process. On the contrary, nowadays the privileges of each userspace application or daemon are minimized, and higher privileges are quite commonly even limited to the system or daemon startup time: after the initialization phase is finished, unnecessary privileges are dropped and cannot be misused by an attacker if the process gets compromised later on. Another common improvement is compartmentalization and isolation of the most vulnerable application parts, such as parsers and renderers, into separate sandboxed processes further reducing attacker's opportunities. As a result of these efforts, modern successful attacks on userspace applications are very complex and require chaining of multiple vulnerabilities in order to reach the desired result. In contrast a single vulnerability directly in the OS kernel provides an attacker with superuser privileges right away.

This thesis focuses on a specific OS kernel, namely the Linux kernel, due to its widespread adoption for mobile and embedded devices: more than 2 billion mobile devices were already running Android OS powered by the Linux kernel [16], many embedded manufacturers build their OSes based on Open Embedded/Yocto [4, 5] projects with the Linux kernel at its base etc. In addition, the Linux kernel is developed fully in the open and any person is able to propose

ideas or contribute the code to the mainline (of course assuming that the Linux kernel maintainers agree with the approach).

Past studies [38, 12] show that the Linux kernel vulnerabilities have remarkably long life times. It takes on average five years between the time when a vulnerability is introduced into the kernel source code and the time it is fixed. Moreover even after it is finally fixed in the mainline Linux kernel, it is very difficult to estimate when the fix is delivered to the end devices: it might take anything between a number of days to a number of years. In addition some older embedded or mobile devices might actually never see these updates, making them perfect attack targets. Thomas et al. [39], in 2015, showed that on average 87.7% of Android devices are exposed to at least 11 known critical vulnerabilities including the ones in the Android OS kernel itself. The time window between the public announcement of kernel vulnerability and its fix (usually pretty fast) in the mainline kernel and the time the fix is propagated to most of the devices is the period of golden opportunity for many attackers: all the details about the vulnerability is known and sometimes even proof-of-concept exploits can be obtained.

As adversaries have been turning towards attacking the Linux kernel itself, security architects, researchers and engineers have been exploring various methods for its better protection. A lot of the effort has been put into improving the overall testing suites for the Linux kernel, various static and dynamic code checkers are also used regularly and automated. New tools, including better commit verification scripts are being developed to assist developers, better code review practices are established etc. However, it is unrealistic to assume that it would ever be possible to have all the bugs in the source code found and fixed even in the well-maintained mainline Linux kernel. The situation is even worse for drivers and other non-mainline code developed by OEMs [38]. Such code might get very limited code review, little testing and be written under harsh time-to-market requirements making it a good place for introducing bugs and vulnerabilities.

An alternative approach that defenders can take is to consider how modern kernel exploits are written and try to eliminate all attack paths that helps exploit writers to succeed. This is exactly the path that the Kernel Self Protection Project (KSPP) [22] has taken. KSPP is an open community of developers and security experts that aims to develop new or adapt existing kernel hardening measures for the mainline Linux kernel.

Figure 4.1 shows how a typical attack on the Linux kernel is carried out. In Step 1 an attacker usually spends time to explore the target kernel offline in order to collect as much information about it as possible. This step is usually performed with high privileges and an attacker has a full access to the kernel memory layout, various debugging and tracing tools etc. Preventing this step is impossible due to the open nature of Linux operating system, but various protection methods can be employed to minimize the value of collected information. For example, Kernel Address Space Layout Randomization (KALSR) [11]

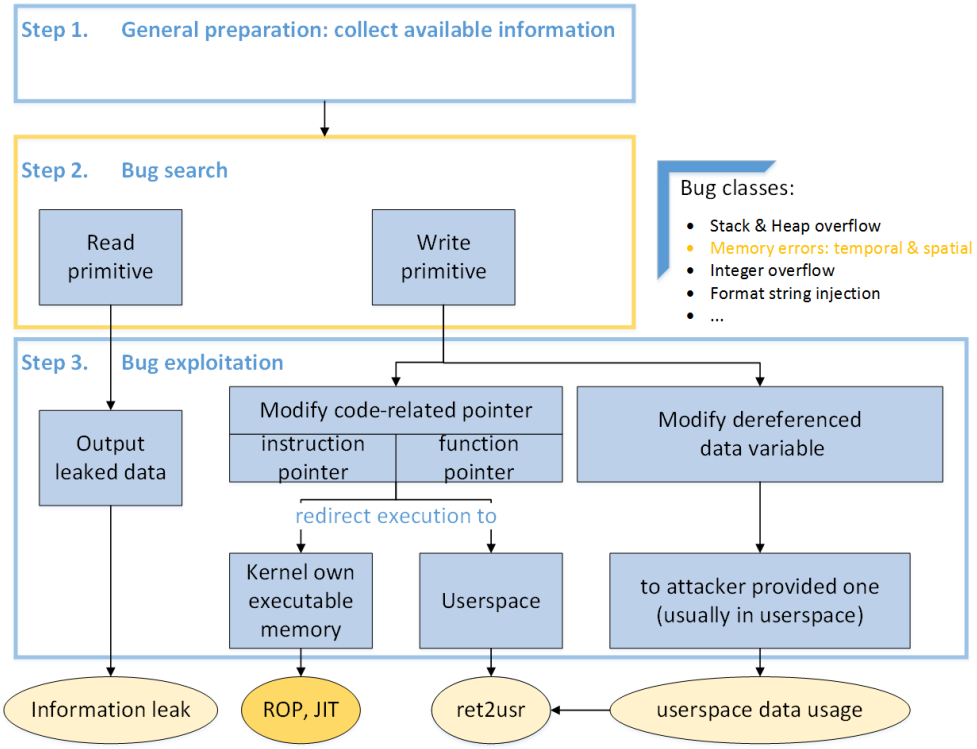


Figure 4.1. High-level overview of a kernel attack

and other techniques such as GRKERNSEC_RANDSTRUCT [3] can make sure that every time a Linux kernel is loaded on the system, the memory locations of many of its components are randomized, and therefore an attacker cannot use the information collected during Step 1 to locate required kernel elements (modules, symbols, structures etc.) during the later steps of a run-time attack.

In Step 2 an attacker needs to find a suitable bug that would allow him to obtain a read or write primitive inside the Linux kernel. There are many bug classes that can be explored for this purpose with the most common ones being memory handling errors (use-after-free, buffer overflows, etc.) and stack/heap overflows

put references

. Preventing Step 2 from happening would be an ideal case for defenders, but in reality it is very difficult and costly to prevent certain classes of bugs from fully happening. Nonetheless a lot of work is being done in this area within the KSPP project and Section 4.2 of this thesis looks into preventing a wide range of memory errors in the Linux kernel.

Last in Step 3 an attacker can finalize the exploitation using the bug(s) found in Step 2. In case of a read primitive, the end result is usually a run-time kernel memory leak that can be either valuable on its own (various sensitive information, such as keys and credentials) or can be used as supporting information for launching further attack steps using another kernel bug. In turn, a write primitive can be used by an attacker to modify either a code-related pointer or

a dereferencing of a kernel data pointer or structure. The code-related pointer can be either directly an instruction pointer (as it is the case with stack overflow attacks) or a function pointer located in places such as function pointer and descriptor tables. All of this would allow an attacker to redirect the execution flow to a desired place. That place can be either the kernel's own executable memory (by building and chaining the available attack gadgets in ROP-like technique or using mechanisms like just-in-time (JIT) compilers) or userspace (various *ret2usr* attacks), where an attacker usually has more control over the memory layout and more mechanisms for continuing the attack and making it persistent over reboots. An attacker can also modify a data pointer (commonly to a structure located in the userspace) that might allow him to perform various data-only attacks. Many different protections can be put in place in order to prevent Step 3 from happening, but they are not generic and depend on the chosen attack path. For example in order to prevent a function pointer overwrite, function pointer tables can be made read-only after the kernel initialization step. Similarly in order to prevent a direct modification of an instruction stack pointer various return address protection techniques, such as stack canaries [15], guard pages [2] etc. can be employed.

Within the KSPP project this thesis has investigated into two important areas of kernel hardening that are highlighted in orange in Figure 4.1). The first one is preventing attackers from misusing the Berkeley Packet Filter (BPF) Just-In-Time (JIT) compiler in order to place the attack payload. The BPF JIT compiler is the only JIT compiler in the Linux kernel that executes programs supplied by unprivileged userspace, making it a very valuable target for the attackers. The second area is the ability to prevent kernel memory errors, such as use-after-frees or buffer overflows, that can greatly affect the overall security of the Linux kernel and help eliminating many past and future vulnerabilities. The subsections below explain each aspect in more details as well as the contribution this thesis makes in addressing these two problem areas.

4.1 Security of Berkeley Packet Filter Just-In-Time compiler

Having an OS kernel mechanism with the ability to execute userspace-supplied code is always very attractive for attackers. The mainline Linux kernel has such a mechanism: the Berkeley Packet Filter (BPF) [34] and the associated Just-In-Time (JIT) compiler, that is basically an in-kernel virtual machine available for unprivileged execution from userspace. Historically this mechanism was introduced to speed up packet filtering on networking sockets, but nowadays it is also used in a wide range of non-networking applications, including trace points [37], syscalls filtering in seccomp [1], Kernel Connection Multiplexer (KCM) [13] etc. There is even a proposal to use BPF for creating a user-programmable Linux Security Module, Landlock [33], that considerably widens the set of BPF use cases.

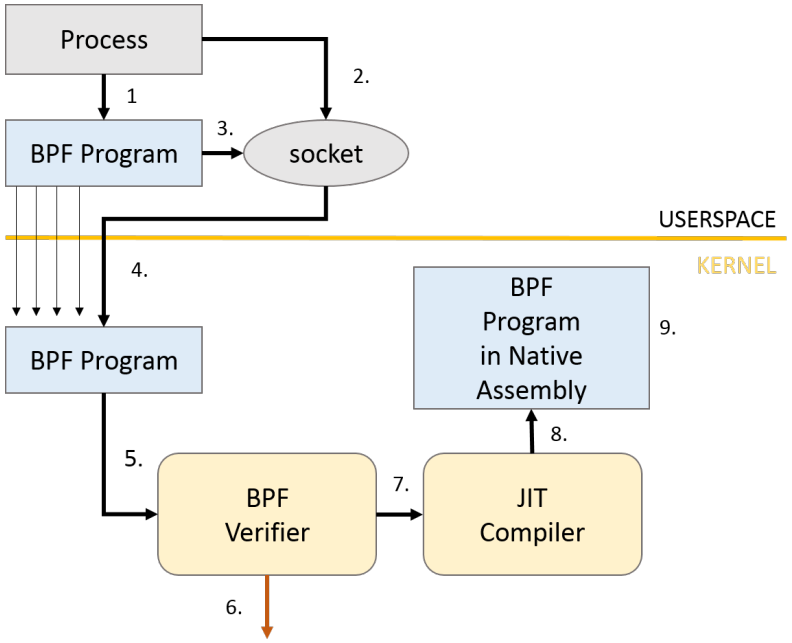


Figure 4.2. Linux kernel Berkeley Packet Filter and JIT compiler (From Publication V)

Remember to either get the copyright permission or modify the figure

The overall flow of how a BPF program is processed by the kernel is shown is shown in figure 4.2. A userspace process expresses its packet filtering algorithm in a form of the BPF program that it wants to attach to a socket. A BPF program is written in the BPF interpreter language that can be executed by BPF in the kernel. When a BPF program is loaded to the kernel, it is first verified by a BPF Verifier component that attempts to check the correctness of supplied program. If the check passes, then the program is attached to the socket and will be executed every time the packet is received for that socket. For non-networking scenarios a BPF program will be executed every time a certain action is performed by its creator, i.e. for the seccomp system calls (syscalls) filtering, a BPF program will be invoked every time a process executes a syscall. In addition in order to further speed up the filtering, the BPF program can be given to the JIT compiler that translates it to native machine assembly language. This is typically enabled for high-performance networking use cases and disabled on a typical desktop system.

The above flow can be abused to load the attacker-supplied payload into the kernel memory using an attack technique called *JIT spraying* [8, 7]. The original proof-of-concept exploit was done back in 2012 [23] and showed that BPF JIT compiler design and implementation are very vulnerable to such attacks. As a result a protective measure was merged in the mainline kernel that attempted to randomize the start memory address within the page where the BPF program gets loaded and will the rest of the page with architecture-specific instructions (aka p̈oisoned zone)̈ that would hang the machine fully if attacker tries to execute them. This made it really hard (with a probability of success of only

about 0.0004%) for an attacker to find supplied payload. However, the root cause of the problem, i.e. an ability to supply the attacker payload in BPF programs using constants, was not fixed at that time, despite the suggestions from some security experts[1].

Publication V of this thesis analyzes the security issues of the BPF JIT compiler and builds a number of successful attacks on the latest (at that time) available mainline 4.4 Linux kernel despite the security fixes done in the past. Our best attack approach, described in Section 4.3 of publication V, analyzes how memory allocation for a BPF program happens and specifically how randomization of the start address is done in the mainline kernel implementation. This analysis discovered that for BPF programs of total size of PAGE_SIZE - 128 - PROGRAM_HEADER_SIZE (size can be controlled by an attacker by adding more or less BPF instructions) the maximum size of the poisoned zone can be calculated in advance (equals to 128 bytes) and it allows an attacker to reliably jump over the poisoned zone and land securely on the BPF program payload. The resulting overall success rate for the attack is 99.6%, which proves that much stronger security measures are required for the BPF JIT compiler in order to provide adequate security against JIT spray attacks. As a result of this work a number of mitigation measures were merged to the mainline kernel that eliminated possibility to perform this type of attacks[2]. It is no longer possible to supply the attacker's payload in BPF programs using the constants due to the blinding process (XORing with a random number) done for each of them prior to its placement in the memory.

4.2 Kernel memory safety

Memory errors can be very dangerous for the security of OS kernel since they can give an attacker the ability to read or write kernel memory, which is normally inaccessible for a userspace process. The cause of these errors is the absence of inherent memory safety in C, which is the primary implementation language of the Linux kernel. Numerous past CVEs (CVE-2014-2851, CVE-2016-4558, CVE-2016-0728, CVE-2014-0196, CVE-2016-8440, CVE-2016-8459, and CVE-2017-7895) as well as vulnerability studies [30, 9] show that memory errors are a big problem for the Linux kernel and needs addressing in order to be more robust against these attacks.

Typically memory errors are divided into two classes:

- **Temporal memory errors** happen when pointers to uninitialized or freed memory are dereferenced. A typical example is a *use-after-free* error when a memory that is dereferenced by the pointer has already been prematurely freed. Another example is a null pointer dereference errors, when an initialized

[1] The Grsecurity kernel security project grsecurity.net

[2] git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=4f3446b

pointer is attempt to being dereferenced.

- **Spatial memory errors** occur when pointers are dereferenced outside the bounds of their intended areas. The most common example here is a buffer overflow when data is written past the end of the memory area allocated for the buffer.

Memory safety has been researched for decades both in academia and industry. Many solutions have been proposed to fully eliminate the root cause of the problem, but they are rarely used in practice due to the significant side effects they bring. The primary issue is usually the performance; most of the developed solutions [19], [29], [28], [24], [20], [41], [40], [26], and [14] incur a significant and non-acceptable run-time performance overhead. Other typical issues are lack of backwards compatibility [25], [17], [6] and the need to make source code changes [25], [17]. In addition, almost all existing mechanisms were developed for the userspace and have not been considered for in-kernel use with the exception of recently developed kCFI [32] and KENALI [36]. However, even these systems have not been developed with the goal of merging them into the mainline Linux kernel and therefore exhibit some design choices that won't be acceptable by the mainline kernel developers[3].

Publication VI of this dissertation takes a deep look at both temporal and spatial memory problems in the Linux kernel and proposes two practical measures that address each of these types of errors.

4.2.1 Preventing temporal memory errors for objects protected by a reference counter

A lifetime of a shared object in the Linux kernel is typically managed using a ***reference counter***. Such counter is incremented every time a reference to an object is taken and decremented when a reference is released. When the counter reaches zero the object can be safely deleted since there are no references to this object anymore (object is not used).

The above presents a simple logic, but practice and history of Linux kernel vulnerabilities (for example CVE-2014-2851, CVE-2016-4558, CVE-2016-0728, CVE-2017-7487 and CVE-2017-8925) shows that reference counting schemes are error prone. If a developer forgets to perform an increment or decrement on some code branch (quite often on a rarely exercised error handling or exception path), it might lead to counter imbalance and as a result incorrect object lifetime handling, i.e. use-after-free situation on the object. This can be misused by attackers in order to construct the efficient and simple kernel exploits. One such example is the exploit based on kernel CVE-2016-0728[4]. The exploit abuses a bug inside kernel keyring facility: forgotten decrement of the reference counter

[3] http://www.openwall.com/lists/kernel-hardening/2017/08/05/1
[4] https://www.exploit-db.com/exploits/39277/

when substituting the session keyring with the same keyring. The problem is even bigger given the fact that the current implementation of reference counters in the Linux mainline kernel allows a counter to overflow upon reaching its maximum value (specific to the architecture).

The first part of the publication VI analyzes how developers have used reference counters in the Linux kernel and contributes to the design of a new reference counter data type `refcount_t` and API that prevents reference counter overflows by its internal design. The new type is specifically designed for the reference counting use only and therefore has a much more limited API compare to the existing generic type, `atomic_t`, used for many other purposes. This allows to reduce the potential development errors when implementing reference counters and improves the code review process. This type and respective API have been accepted into the mainline Linux kernel and are becoming a de-facto standard for implementing reference counters. One of the biggest parts of the work was an overall kernel-wide analysis and conversion of existing reference counters to the newly available type and API that resulted into more than 170 accepted patches

update exact number since it is changing all the time

. The analysis was done using a static analyzer, Coccinelle [10], integrated into the Linux Kernel build system (KBuild). A set of code patterns was developed for this purpose that identify reference counters based on their behavior. These code patterns are explained in more details in Section 4.1 of the Publication VI. After the automatic analysis has been completed, each case was manually analyzed and a respective patch created.

4.2.2 Preventing out-of-bounds memory accesses

As was already mentioned in Section 4.2 none of the existing run-time mechanisms to handle memory access errors are directly applicable to the mainline Linux kernel. However other mechanisms are commonly used for the debugging purposes. KASAN [21], integrated in the mainline Linux kernel, is perhaps the most known of them all. It allows to detect a wide range of memory errors, including spatial ones. However its performance overhead is unsuitable for most production use cases. Another examples include popular Valgrind [27] and AddressSanitizer [35] tools.

Recently Intel has released a new technology, Intel Memory Protection Extensions (MPX) [31], for an efficient run-time prevention of spatial memory errors that is targeting a wide range of end devices from desktops to mobile and embedded processors. As many related solutions, MPX does it by determining (either during compile time or during the run-time) the correct bounds for each used pointer and storing this data in the separated metadata storage. Pointer bounds are enforced by MPX instrumentation using added hardware instructions, essentially by checking memory addresses before dereferencing pointers. The checks are done against bounds set by compiler instrumentation, either

statically based on data structure sizes, or during execution by instrumented allocators. The above software support is implemented both in the Linux kernel and the GCC compiler, but until recently, this software support only worked for user-space applications.

The second part of the publication VI analyzes the software implementation of Intel MPX as well as various challenges of using MPX for the Linux kernel itself. The major challenge was a high memory overhead incurred by the mechanism how MPX stores the bounds for the pointers. If such mechanism is directly applied to the Linux kernel, it would result in more than 500% increase for the kernel base memory, which is unacceptable. The on-demand memory allocation mechanism for the userspace, used by MPX in attempt to minimize its memory consumption, is also not possible to implement reliably for the Linux kernel itself. Other challenges included an inability to use the MPX initialization code created for the userspace, userspace function wrappers, as well as the GCC compiler support directly for the kernel code. As a result Publication VI proposes a new mechanism, *MPX for Kernel (MPXK)*, which is an adaptation of Intel MPX for in-kernel use where all the above described challenges has been solved. MPXK does not use the costly memory storage mechanism of MPX, but instead re-uses the kernel memory management metadata created by `kmalloc`-based allocators. This results in a small memory footprint, but has some limitations explained in details in Sections 5.1 and 6.1 of Publication VI. MPXK also implements all the missing GCC compiler instrumentation in a form of a new GCC plugin, as well as does in-kernel initialization of MPX and function wrappers to pass the bounds metadata.

4.3 Discussion

In order to perform a high-level evaluation of the solutions proposed in this chapter and their impact on the researched area, it is important to start from the original objectives defined in Section 1.2: security, practical deployability and usability. A more formal and detailed evaluation of solutions proposed in this chapter can be found in the evaluation sections of publications V and VI.

The attacks described in Section 4.1 built a strong case for the mainline acceptance of BPF JIT blinding protections that considerably improved security of Linux kernel BPF/JIT. Originally this attack was possible because the blinding solution developed by Grsecurity [18] was considered as an unnecessary complication to the BPF JIT functionality impacting code maintainability and performance. And since the only existing practical 2012 proof-of-concept exploit [23] could be stopped with a much simpler measures (start address randomization and poisoning zone), the constant blinding technique was not implemented in the mainline kernel until the work within the KSSP project on BPF/JIT has started.

The measures to harden Linux kernel reference counters, described in Sec-

tion 4.2.1, aim to significantly decrease hard-to-find developers mistakes on reference counter usage and therefore prevent resulting use-after-free situations on shared kernel objects. If such measures would be in place in past, they would have prevented many kernel vulnerabilities mentioned in Section 4.2.1. The newly introduced `refcount_t` type and API guarantee that a reference counter cannot overflow under all conditions, and our kernel-wide conversion of reference counters to this newly available type (175 from 220) provides a good initial code coverage with more and more conversions added in each kernel release. While the overall effectiveness of this measure would be only possible to judge with time, the proposed solution is also practically deployable and usable: it exhibits acceptable performance overhead for most of workloads, can be applied on case-by-case basis and it has been taken into use by its main intended users, kernel developers and maintainers.

The MPXK mechanism, described in Section 4.2.2, aims to protect the Linux kernel from all spatial memory errors and therefore prevent a wide class of attacks caused by them. MPXK ensures that pointers to the objects with known bounds cannot be dereferenced outside of these bounds. However in some cases the correct bounds cannot be reliably determined and therefore MPXK has a number of limitations connected with usage of legacy code, type of memory allocation and some cases of direct pointer manipulations. The practical deployability of MPXK is judged based on a small performance overhead and an ability to apply the protection only in required places (with a compilation unit granularity) compared to the system-wide solutions like KASAN. MPXK does not require any source code changes and it is fully integrated into the Kernel build infrastructure making it easy for kernel developers to enable protection where required.

5. Future Outlook and Conclusions

Is it better to have discussion section at the end of each subsection with potential outlook into the future etc.? Then this section can be just future work summary and conclusions.

Outlook for mobile and platform security: IoT, automation, machine learning (since we have a lot of data).

References

[1] SECure COMPuting with filters. *www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt*, 2016.

[2] Grsecurity wiki: Prevent kernel stack overflows. *https://en.wikibooks.org/wiki/Grsecurity/Appendix/Grsecurity_and_PaX_Configuration_Options#Prevent_kernel_stack_overflows*, 2017.

[3] Grsecurity wiki: Randomize layout of sensitive kernel structures. *https://en.wikibooks.org/wiki/Grsecurity/Appendix/Grsecurity_and_PaX_Configuration_Options#Randomize_layout_of_sensitive_kernel_structures*, 2017.

[4] Openembedded project, 2017.

[5] Yocto project, 2017.

[6] Todd M. Austin, Scott E. Breach, Gurindar S. Sohi, Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation - PLDI '94*, 29(6):290–301, 1994.

[7] Piotr Bania. JIT spraying and mitigations. *arXiv preprint arXiv:1009.1038*, 2010.

[8] Dion Blazakis. Interpreter Exploitation: Pointer Inference and JIT Spraying. *www.semantiscope.com/research/BHDC2010/BHDC-2010-Paper.pdf*, 2010.

[9] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M Frans Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, page 5. ACM, 2011.

[10] Coccinelle Project. http://coccinelle.lip6.fr/, 2017.

[11] Kees Cook. Kernel address space layout randomization. *https://lwn.net/Articles/569635/*, 2016.

[12] Kees Cook. Status of the Kernel Self Protection Project. *www.outflux.net/slides/2016/lss/kspp.pdf*, 2016.

[13] Jonathan Corbet. The kernel connection multiplexer. *lwn.net/Articles/657999/*, 2015.

[14] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for c with very low overhead. In *Proceedings of the 28th international conference on Software engineering*, pages 162–171. ACM, 2006.

[15] Jake Edge. "Strong" stack protection for GCC. *https://lwn.net/Articles/584225/*, 2014.

[16] Google I/O 2017 Highlights. https://www.slideshare.net/AppInventiv/google-io-2017-highlights, 2017.

[17] Dan Grossman, Michael Hicks, Trevor Jim, and Greg Morrisett. Cyclone: A type-safe dialect of c. *C/C++ Users Journal*, 23(1):112–139, 2005.

[18] grsecurity. https://grsecurity.net, 2017.

[19] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *In Proc. of the Winter 1992 USENIX Conference*, pages 125–138, 1991.

[20] Richard WM Jones and Paul HJ Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Proceedings of the 3rd International Workshop on Automatic Debugging; 1997 (AADEBUG-97)*, number 001, pages 13–26. Linköping University Electronic Press, 1997.

[21] The Kernel Address Sanitizer (KASAN). www.kernel.org/doc/html/v4.10/dev-tools/kasan.html, 2017.

[22] Kernel Self Protection Project wiki. http://www.kernsec.org/wiki/index.php/Kernel_Self_Protection_Project, 2017.

[23] Keegan McAllister. Attacking hardened Linux systems with kernel JIT spraying. *mainisusuallyafunction.blogspot.de/2012/11/attacking-hardened-linux-systems-with.html*, 2012.

[24] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. 44(6):245–258, 2009.

[25] George C Necula, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy code. In *ACM SIGPLAN Notices*, volume 37, pages 128–139. ACM, 2002.

[26] Nicholas Nethercote and Jeremy Fitzhardinge. Bounds-Checking Entire Programs without Recompiling. *Proceedings of the Second Workshop on Semantics Program Analysis and Computing Environments for Memory Management SPACE 2004*, 2004.

[27] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 42, pages 89–100. ACM, 2007.

[28] Harish Patil and Charles Fischer. Low-cost, concurrent checking of pointer and array accesses in c programs. *Softw., Pract. Exper.*, 27(1):87–110, 1997.

[29] Harish Patil and Charles N Fischer. Efficient run-time monitoring using shadow processing. In *AADEBUG*, volume 95, pages 1–14, 1995.

[30] Supriya Raheja, Geetika Munjal, et al. Analysis of linux kernel vulnerabilities. *Indian Journal of Science and Technology*, 9(48), 2016.

[31] Ramu Ramakesavan, Dan Zimmerman, and Pavithra Singaravelu. Intel Memory Protection Extensions (Intel MPX) enabling guide, 2015.

[32] Sandro Rigo, Michalis Polychronakis, and Vasileios P Kemerlis. DROP THE ROP Fine-grained Control-flow Integrity for the Linux Kernel, 2017.

[33] Mickaël Salaün. Landlock lsm, 2016.

[34] Jay Schulist et al. Linux Socket Filtering aka Berkeley Packet Filter (BPF). *www.kernel.org/doc/Documentation/networking/filter.txt*, 2016.

[35] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

[36] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing Kernel Security Invariants with Data Flow Integrity. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2016.

[37] Alexei Starovoitov. Tracing: attach eBPF programs to kprobes. `lwn.net/Articles/636976/`, 2015.

[38] Jeff Vander Stoep. Android: protecting the kernel, 2016.

[39] Daniel R. Thomas, Alastair R. Beresford, and Andrew Rice. Security metrics for the android ecosystem. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '15, pages 87–98, New York, NY, USA, 2015. ACM.

[40] Wei Xu, Daniel C DuVarney, and R Sekar. An Efficient and Backwards-compatible Transformation to Ensure Memory Safety of C Programs. *SIGSOFT Softw. Eng. Notes*, 29(6):117–126, 2004.

[41] Suan Hsi Yong and Susan Horwitz. Protecting c programs from attacks via invalid pointer dereferences. In *ACM SIGSOFT Software Engineering Notes*, volume 28, pages 307–316. ACM, 2003.

Errata

Publication I

Kostiainen, Kari and Reshetova, Elena and Ekberg, Jan-Erik and Asokan, N.. Old, new, borrowed, blue: a perspective on the evolution of mobile platform security architectures. In *Proceedings of the First ACM Conference on Data and Application Security and Privacy*, Location, pages, and other detailed information, Month 2011.

Old, New, Borrowed, Blue – A Perspective on the Evolution of Mobile Platform Security Architectures

Kari Kostiainen
Nokia Research Center
kari.ti.kostiainen@nokia.com

Elena Reshetova
Nokia
elena.reshetova@nokia.com

Jan-Erik Ekberg
Nokia Research Center
jan-erik.ekberg@nokia.com

N. Asokan
Nokia Research Center
n.asokan@nokia.com

ABSTRACT

The recent dramatic increase in the popularity of "smartphones" has led to increased interest in smartphone security research. From the perspective of a security researcher the noteworthy attributes of a modern smartphone are the ability to install new applications, possibility to access Internet and presence of private or sensitive information such as messages or location. These attributes are also present in a large class of more traditional "feature phones." Mobile platform security architectures in these types of devices have seen a much larger scale of deployment compared to platform security architectures designed for PC platforms. In this paper we start by describing the business, regulatory and end-user requirements which paved the way for this widespread deployment of mobile platform security architectures. We briefly describe typical hardware-based security mechanisms that provide the foundation for mobile platform security. We then describe and compare the currently most prominent open mobile platform security architectures and conclude that many features introduced recently are borrowed, or adapted with a twist, from older platform security architectures. Finally, we identify a number of open problems in designing effective mobile platform security.

Categories and Subject Descriptors

D.4 [**Operating Systems**]: Security and Protection

General Terms

Security, Design

1. INTRODUCTION

In the past few years, there has been a dramatic increase in the popularity of the category of mobile phones commonly known as "smartphones." Consequently there is increased interest in the security and privacy research community in "smartphone security". What exactly constitutes a "smartphone" is a matter of debate [16]. But from the perspective of a security researcher, some attributes of smartphones stand out. One is the ability to extend the functionality of phone by incorporating new software components. At the moment, this takes the form of installing new applications. Another is the ability to access (and be accessed from) the Internet. A third is the presence of private or sensitive information like personal messages, location etc.

These security-relevant attributes lead us to two important observations. First, these attributes are also present in a large class of mobile phones commonly known as "feature phones". For example, Java Platform Micro Edition (Java ME), which makes it possible for application developers to develop and deploy Java midlets to mobile devices, is reportedly present on over three billion phones [23]. Therefore, we argue that instead of focusing on "smartphone security", security researchers should study mobile platform security more generally.

Second, these attributes are instantly recognizable as characteristics of any personal computer (PC) platform. PC platforms started out as open systems with no platform security schemes in place. Even today, security mechanisms in PC platforms are based primarily on perimeter control, like firewalls, and reactive mechanisms like anti-virus tools. Although various platform security architectures (such as Security-Enhanced Linux [17]) have been designed and implemented, none has seen widespread deployment.

In contrast all significant mobile phone platforms have widely deployed platform security schemes. The primary reason for this is how the business, regulatory, and end-user requirements on mobile phones have shaped the evolution of mobile platform security over the last decade or so.

In this paper, we begin by taking a brief look at the motivation and background for mobile platform security and the requirements they implied. Then we describe the types of hardware-security mechanisms that provide the foundation for mobile platform security. After that we describe and compare the currently most prominent open mobile platform security architectures. Finally, we identify a number of open problems in designing effective mobile platform security.

2. BACKGROUND AND MOTIVATION

In contrast to PC platforms, mobile phones began as closed systems with limited functionality. Right from the beginning different stakeholders had certain clear security require-

CODASPY'11, February 21–23, 2011, San Antonio, Texas, USA.

ments for mobile devices. For example, in many parts of the world, several *mobile network operators* provide subsidized mobile phones to their customers in return to commitment to a specified contract period. They require mobile phones to incorporate mechanisms to enforce *subsidy locks*, to prevent a customer who received a subsidized mobile phone from changing operators before the subsidy contract was ended. *Regulators*, like the Federal Communication Commission in the United States, are interested in aspects affecting the public good. For example, during manufacture a mobile phone undergoes a configuration process where the parameters for radio frequency (RF) operations (like transmission power) are calibrated and stored. Regulators want to ensure *secure storage* of such RF parameters for end-user safety. Additionally, if a malicious user could manipulate these parameters he could gain unfair advantage (bigger communication bandwidth than was intended for him) or disrupt communications for other users. Regulators are also interested in theft-deterrent mechanisms such as the means to track stolen devices.

Some of these requirements made their way into standards specifications. The European Telecommunications Standards Institute (ETSI), the body that originally specified the Global System for Mobile Communications (GSM) standard had representatives from mobile device manufacturers, mobile network operators as well as regulators. In the early 1990s, a version of the ETSI recommendation on "security aspects" recommended protection of the IMEI (International Mobile Equipment Identifier, a unique code for a specific device) and the IMSI (International Mobile Subscriber Identifier, a unique code for each subscription) by specifying that "Both IMSI and IMEI require physical protection. Physical protection means that manufacturers shall take necessary and sufficient measures to ensure the programming and mechanical security of the IMEI" [10]. Immutability of IMEI and IMSI is essential for enforcing subsidy lock. Immutability of IMEI also serves as a theft-deterrent mechanism.

Ten years later, by the time the first "smartphone" (the Nokia 9210) appeared, the importance of these basic requirements grew. A subsequent version of the same ETSI specification re-iterated the requirement that "The IMEI shall not be changed after the ME (mobile equipment) final production process. It shall resist tampering, i.e. manipulation and change, by any means (e.g. physical, electrical and software)" [11]. It also implied that compliance to this requirement would be needed for type approval by stating that "This requirement is valid for new GSM ... MEs type approved after 1st June 2002."

In addition to operator and regulator requirements, it was also evident that *end-users* had come to expect a certain level of predictability and reliability from mobile phones. Unlike in PC platforms, where malfunctioning system software or malicious applications are usually merely a nuisance to the user, in mobile domain such software can cause considerable harm to him, e.g. in the form of monetary losses due to an increased phone bill. To retain that level of end-user trust while opening up mobile phone platforms for application installation and Internet connectivity, the platforms needed to support appropriate platform security schemes.

Mobile platform vendors responded to these operator, regulator and end-user needs by developing "hardware-security mechanisms" and by integrating "platform security architectures" to the mobile operating systems and application platforms. In the next sections of this paper, we describe typical hardware-security mechanisms used in mobile phones and compare widely deployed open platform security architectures.

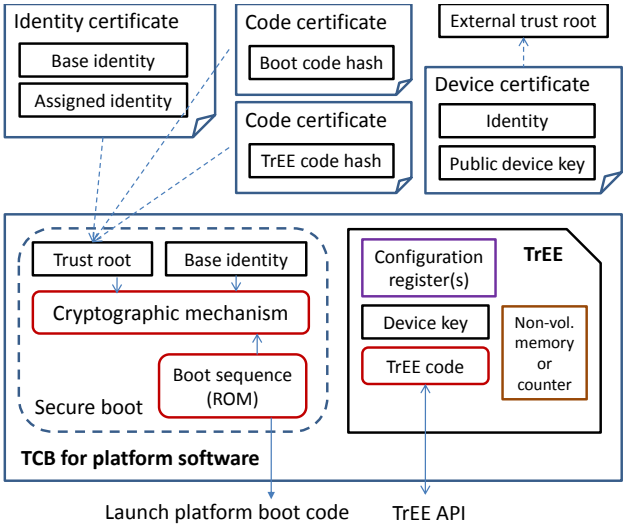


Figure 1: Common hardware-security mechanisms used in mobile devices.

3. HARDWARE-SECURITY MECHANISMS

The augmentation of device hardware to support operating system security is by no means a new invention. As early as in 1971, Lampson examined the concept of protection domains [15] and e.g. the Cambridge CAP computer [37] developed in the 1970s provided hardware support for such a feature. The first standardization efforts for hardware-assisted security took place in late 1990s when the Trusted Computing Platform Alliance (TCPA) [25], a predecessor of current Trusted Computing Group (TCG) [34], defined a hardware-based security element for PCs. Roughly at the same time, the first large scale deployments of hardware-assisted security started when mobile device manufacturers added hardware-based integrity protection of booted system software. The mechanism was originally designed as a safety feature rather than a security one, but in the context of mobile devices its primary purpose was to prevent software replacement in the field.

In this section we describe common hardware-security mechanisms used in mobile devices today (see Figure 1), and explain the rationales of mobile device manufacturers for introducing such security mechanisms. Then we briefly list some existing and standardized hardware-security architectures in the mobile domain that implement some or all of the explored mechanisms.

3.1 Base identity and trust root

For typical mobile devices, two pieces of immutable information are needed in hardware. First, virtually all devices need at least one *base identity* that uniquely identifies the device hardware. The base identity can be the IMEI of the device or any other (statistically) unique identifier. Immutability of the device identifier can be achieved e.g. by storing the value in a read-only memory (ROM) during the

manufacturing of the application-specific integrated circuit (ASIC).

Second, mobile devices need to authenticate external information (see e.g. assigned identities and secure boot in Sections 3.2 and 3.3). A pre-condition for external information authentication is that another piece of immutable information, a *trust root*, is stored on the device hardware during manufacturing. A typical trust root is a hash of the device manufacturer public key [32, 33], and thus the trust root is usually shared within a family of manufactured devices, unlike the device-specific immutable identity which is typically unique.

3.2 Assigned identities

Mobile devices usually need more than one identity, e.g. for MAC addresses of all supported radio interfaces. The ability to assign additional device identities after the ASIC manufacturing, rather than to fixing them into ROM at the time of manufacturing, gives more flexibility to the device manufacturers. The combination of a trust root and at least one base identity gives the device manufacturer or integrator the possibility to assign arbitrarily many other identities to a chosen device.

One way to achieve this is to issue an *identity certificate* for the device. This certificate is signed with respect to the device trust root and it binds the an *assigned identity* to the base identity. Additionally, the device hardware must be enhanced with a *cryptographic mechanism* that can verify the identity certificate (or a certification chain). The operational integrity of the cryptographic mechanism implementation must be trustworthy, and thus such algorithms, say RSA implementation, are often part of the ROM on the ASIC, where they can be deployed during the boot sequence at a stage where no untrusted code has yet been run.

3.3 Secure boot

Due to regulatory, operator and end-user requirements some mobile device manufacturers assert the integrity of system software during device boot-up and stop the boot process if system software has been modified. This process is called *secure boot*. To implement secure boot, a typical approach is to make the beginning of the *boot sequence* immutable, e.g. by the virtue that it resides in ROM, and that the processor unconditionally starts executing from that memory area. The device manufacturer can issue a set of *code certificates* that contain *boot code hashes* and are signed with respect to the device trust root. Again, the hardware must be enhanced with a cryptographic mechanism that validates the first executed system component (e.g. boot loader) before it further validates and executes the next one (e.g. OS kernel). If any of the validation steps fail, the boot process aborts.

This "measure-before-execute" principle can be iterated as many times as necessary to include not just the initially launched OS kernel (or hypervisor) binary, but also any other code or configuration data whose integrity needs to be validated. Secure boot based on code signing does not necessarily imply that only a single code set-up can be booted on a given device. The launched image may be one of several certified alternatives, depending on user selection (see Section 5) or other external or internal context.

3.4 Trusted Execution Environment

All the hardware-security mechanisms listed so far have been integrity-related, and implementation of these mechanisms does not require storage of or operation on secrets. There are, however, security services and use cases (see Section 3.7) that require secure storage and isolated execution.

A way to construct an isolated execution environment is to validate the Trusted Computing Base (TCB) using secure boot, since TCB by definition is isolated from the rest of the system. This approach may suffice, especially in cases where the TCB is small, like a hypervisor. For complete operating systems (kernels) this is not a viable approach, since these are so large that implementation bugs that enable software-attacks against TCB are inevitable.

When dealing with confidential information, the attack model also changes. When attacking integrity features, isolation must be broken at every boot, whereas a breach against confidentiality needs to succeed only once and the secret falls into the hands of the attacker. Thus, where confidentiality is concerned, at least simple hardware attacks like memory-bus monitoring or side-channel attacks must be considered as plausible attack vectors in addition to software-based attacks.

To overcome the security problems of TCB that consists of entire OS kernel, mobile device manufacturers have enhanced their ASICs with support for *Trusted Execution Environment* (TrEE). A typical TrEE includes secure storage for a (statistically) unique *device key* and an execution environment in which small pieces of code (*TrEE code*) can be executed isolated from the rest of the system. The isolated execution environment is typically based on on-chip memory to alleviate memory-bus attacks. Combined with the secure boot features, trust roots and identities described above, this set effectively can become a minimal *TCB for platform software* to be leveraged by the booted operating system or platform software on the device.

Like with any TCB, since the TrEE will contain and provide access to secrets, there must be some assurance that code that is run inside it will not, as part of its operation, reveal the secrets to outside, either accidentally or by malice. This assurance can be achieved either by code-signing (code certificates that contain TrEE code hashes) or by constructing the TrEE such that any code run in it gets only indirect access to the confidential data (e.g. a secret key can be applied to a cryptographic algorithm, but it is never directly accessible to TrEE code). The same applies to runtime protected data that may be stored in TrEE memory. TrEE typically provides an API for loading executed code.

3.5 Configuration registers

Additionally, TrEEs often support *configuration registers*. These registers can be used to store measurements from the (possibly validated) booted software or an aggregate of the non-validated code the TCB executes over time. Also hardware configuration options, configuration file contents or user inputs can be stored on these registers. The integrity of configuration registers must be guaranteed, but it is not persistent across boot cycles. The information contained in such registers can be used in two ways. First, code run inside the TrEE can adjust their logic based on the current system state, or e.g. user input received at boot when no untrusted software still was running. Secondly, the state of the system can be attested to a remote party (see Section 3.7).

3.6 Authenticated boot

A boot process in which (1) the measure-before-execute principle is followed during boot without certificate validation nor possible boot termination, and (2) all intermediary measurement results are added to configuration registers (without the possibility of data rollback), is often called *authenticated boot*. The aggregate information from the registers will uniquely identify the system state up to the instance where the first piece of untrusted code is launched. Put another way, if no untrusted code was launched, this fact can later be attested to external parties without the need for certificate checking or halting during boot.

3.7 Sealing and attestation

A persistent device key initialized during ASIC manufacturing and only accessible within the TrEE, can be used for various security mechanisms. First, with this key, or any derivation from it, the TrEE can locally encrypt data, and then give the result to an unprotected domain, such as external memory, for persistent storage. This mechanism is often called *sealing*.

If the device key is usable for public key cryptography, then an external certification authority (CA, typically run by the device manufacturer) may issue a *device certificate* that binds the public part of the device key to any of the device identities (base or assigned). The certification process can take place e.g. during device manufacturing, when the CA can still trust the integrity of the public part of the device key.

The *public device key* together with the device certificate can be used to set up an authenticated and confidential communication channel from an external party into the TrEE, e.g. for provisioning or data migration purposes. Also, the TrEE can by means of a signed statement to a third party remotely attest any state information inside it, e.g. the measurements stored to configuration registers from an authenticated boot sequence.

3.8 Statefulness

Many services in which the user can be considered an adversary, such as digital rights management, device lock PIN retry control or micropayment protocols, need reliable rollback protection of system state. During one device boot cycle, rollback protection is implementable using the configuration registers described in Section 3.5. To implement *off-line* rollback protection, stored data needs to be bound to reliable time information within the TrEE. Straightforward ways to implement this is to include either a monotonic *counter* or *non-volatile memory* in the TrEE.[1]

3.9 Hardware-security architectures

Most of the hardware-security mechanisms described above have seen widespread deployment in proprietary mobile *hardware-security architectures*, such as ARM TrustZone [3, 4] or M-Shield [32]. TrustZone architecture augments the processing core to provide a new set of processing (register) contexts, as well as memory management unit (MMU) and direct memory access (DMA) security integration. ASICs with TrustZone architecture support secure boot, and can also be populated with isolated RAM and ROM residing within the ASIC package to provide memories resistant to simple hardware attacks. For confidential information, trust roots and identifiers some amount of chip-specific "write once, read many times" persistent memory (typically implemented with E-fuses) is also available. Thus TrustZone provides secure boot, integrity protected trust roots and device identifiers, confidential device keys and isolated execution for TrEE code, but typically lacks secure non-volatile memory and counters, and thus rollback protection must be set up with external means.

On the PC side, the most widely deployed hardware-security architecture is the Trusted Platform Module (TPM) [35] defined by Trusted Computing Group (TCG). TPM by definition is a self-contained, stand-alone secure element. TPM does not provide secure boot, but combined with an associated processor feature, the Dynamic Root of Trust for Measurement (DRTM), it can be used to set up a limited TrEE [19] within the logic confines of an Intel VTx or AMD processors. In this context TPM has trust roots, device secrets, device certificate capability, counters, i.e. all features listed above.

TCG has also defined a standard called Mobile Trusted Module (MTM) [9] for mobile devices. Unlike TPM, MTM is an interface specification that can be implemented using various means, e.g. using TrustZone. MTM specification supports most TPM features, with a few mobile-domain specific additions.

4. OPEN MOBILE PLATFORMS

In this section we briefly describe the prominent open mobile operating systems and application platforms: Symbian, Java ME, Android and MeeGo. We focus on mobile platforms for which reference implementations and security architectures are publicly available, and exclude other popular mobile platforms, such as iPhone, Windows Phone 7 and Blackberry, that are open to third-party application development but the internals of which are not public.

4.1 Symbian

Symbian is an evolution from EPOC operating system used in Psion devices in 1990s. In Symbian most of the operating system services, such as file system and networking, are implemented as user-space system servers. Communication between system servers and user applications takes place via built-in interprocess communication (IPC) framework [28]. Symbian platform security architecture was added in 2005 and at the time Symbian was the first smartphone OS to incorporate a platform security architecture. Symbian is currently the most used smartphone OS [12] and primarily used in Nokia devices. Application development in Symbian is done in C++ and using Qt framework. Symbian operating system supports three types of executables: UI applications, background servers and libraries.

Symbian developers define two configuration files for each application: a project definition file (MMP file) defines project settings, such as the identifier of the application and source code files and libraries used, and a packaging file (PKG file) controls how an installation package in constructed.

In the Symbian platform security architecture, access to

[1]Many existing mobile devices support neither non-volatile secure memory nor secure counters (see e.g. [30] for rationale). In some cases the need for *local* rollback protection can be mitigated; e.g. when on-line server communication is an unconditional requirement of the use case at hand, the rollback-protection can be server-assisted, i.e. the server provides authenticated time information.

protected APIs is controlled using a finite set of permissions which are called "capabilities". Applications that access protected APIs must be signed by a central trusted authority (SymbianSigned). During the signing process the trusted authority checks that the application conforms to publication criteria and assigns a globally unique application identifier (Secure Identifier, SID) from a *protected range* of application identifiers. The authority maintains a mapping between the issued identifiers and identities of the software issuers. For most applications the developer identity verification is based on simple online registration (nominal fee of 1 euro). For applications that require restricted capabilities, the application developer must purchase a publisher identity certificate (200 USD per year). Nokia Ovi Store is the primary distribution channel for Symbian applications.

Symbian applications that do not require access to protected APIs can be self-signed and distributed via other channels. In such a case, the developer picks the application identifier from an *unprotected range*.

4.2 Java ME

Java Micro Edition (Java ME) is an application platform supported by various devices from embedded devices to mobile phones and set-top boxes. Java ME platform consists of device "configurations" that define the used Java virtual machine and the core APIs, and "profiles" that define additional APIs for building complete applications. Mobile phones typically support Connected, Limited Device Configuration (CLDC) and Mobile Information Device Profile (MIDP). Java ME is the most widely supported third-party application development platform for feature phones with over 3 billion devices deployed [23]. However, many latest high-end smartphone platforms, including Android, iPhone and MeeGo, do not support Java ME.

The current MIDP standard [21] supports only standalone UI applications, or *midlets*.[2] Midlets are packaged into JAR files before deployment. Applications specific attributes, such as name, version, vendor and requested permissions, are shipped in a Java Desriptor (JAD) file or manifest file.

In Java ME platform security architecture access to protected APIs is controlled with permissions. Application signing binds the application to a *protection domain* according to a local security policy. The policy defines the permissions that applications of each protection domain can have. Typically mobile phones have four predefined protection domains: device manufacturer domain, network operator domain, and domains for identified and unidentified third-party applications.

4.3 Android

Android is a Linux-based smartphone OS developed by Google. Android was released in 2008 and currently it is the second most used smartphone operating system [12]. Application development in Android is primarily done in Java, although applications can include native components as well. Each Android application runs in a separate Dalvik virtual machine in its own process context. Android provides an IPC framework for communication between Java applications.

[2]Next MIDP version [22] adds support for background midlets, shared libraries and inter-midlet communication. We exclude analysis of these features from our discussion, because this version is not yet widely supported.

In Android platform security architecture access to protected APIs is controlled with permissions. Android applications are distributed as Android packages. A manifest file inside the package defines the permissions requested by the application and permissions required to use the services (IPC APIs) offered by the application. Android applications must be signed before installation to the device. Most third-party Android applications can be self-signed (applications accessing system APIs must be signed by Google). Android Market is the primary distribution channel for Android applications. Publishing an application requires a registration fee of 25 USD.

4.4 MeeGo

MeeGo is an upcoming Linux-based mobile platform developed jointly by Intel and Nokia. MeeGo is an evolution from Nokia's Maemo platform and Intel's Moblin OS. Compared to Android, MeeGo is much closer to a standard Linux distribution. Application development is primary done in native C/C++ and using Qt framework. MeeGo supports IPC between applications via standard Unix sockets and with Desktop Bus (D-Bus) framework [6].

MeeGo provides a new platform security architecture called Mobile Simplified Security Framework (MSSF) [14, 13]. MSSF is an evolution from Maemo 6 platform security solution, which was initially developed by Nokia. In this paper we will concentrate on the latest design of MSSF framework [20]. In MSSF access to sensitive APIs and files on the device can be controlled using both traditional Linux access control rules and permissions that are called "resource tokens".

MeeGo applications may be installed from various software sources. The notion of a "software source" is abstract and can represent a different range of entities starting from central software repository, such as Nokia Ovi Store, to single developers. Individual software sources are part of a tree-like structure, and a *trust level* is associated with each software source in hierarchical manner.

MeeGo applications are distributed as RPM packages. Each package must be signed by the software source. MeeGo devices have a local list of known software sources and their public keys. A local security policy on a MeeGo device defines *trust levels* for known software sources and permissions that each software source is allowed to grant. The local security policy can either be defined by a manufacturer, operator or even a device user, when a device booted in "developer mode" (see Section 5).

5. OPERATING SYSTEM BOOTSTRAPPING

In this section we start a comparative security analysis of the open mobile platforms described in the previous section. We begin our analysis be comparing different operating system validation and bootstrapping approaches that are used in these platforms.

Symbian. Most Symbian devices support secure boot, with hardware-security architectures like ARM TrustZone, as described in Section 3. Thus, with Symbian devices developers cannot boot their own custom kernels.

Android. In Android, the bootstrap issue is up to the device manufacturer. There is little information available about the different bootstrapping schemes chosen by different Android device manufacturers, but at least in principle developers can update devices with custom kernels [36].

MeeGo. Also in MeeGo different device manufacturers

may implement different OS bootstrapping strategies. Nokia MeeGo devices can support a dual boot approach in which the device can be booted to *normal mode* with official OS kernel image provided by the device manufacturer or to *developer mode* with custom kernel provided by any developer [26]. Integrity of each component of the boot sequence, starting from the bootloader, is verified using boot certificates (see Section 3.3 for more details). However, unlike in usual secure boot, if the integrity verification of operating system image fails, the boot process is not halted, but the user is notified and asked permission to continue the boot. If the user decides to continue, this information is stored in a configuration register inside TrEE.

Later, when the OS requests access to certain device secrets, such as digital rights management keys, this access is prevented (inside TrEE) if the device was booted to developer mode. Both modes allow user space applications to utilize cryptographic services provided by the TrEE. The keys for these services are derived from the device key, and the derivation process includes information about the device mode. This guarantees that the content, encrypted in the normal mode, can not be decrypted in the developer mode.

6. PLATFORM SECURITY ARCHITECTURE COMPARISON

Next, we compare the platform security architectures of mobile platforms described in Section 4. The key differences and similarities are summarized in Table 1.

6.1 Application identification

Symbian. During application installation and at runtime each executable (application or background server) has two identifiers: Secure Identifier (SID) uniquely identifies the executable and Vendor Identifier (VID) identifies the software vendor. These identifiers are typically assigned by the central trusted authority; for self-signed applications the developer may pick any SID from the unprotected range. Libraries inherit SID and VID from the executable that loads them.

Java ME. During installation, a midlet is identified based on installation package signature key and midlet attributes, such as package name. Midlets are standalone applications that do not communicate with each other, and thus runtime code identification is not applicable to Java ME.

Android. During installation Android applications are identified based on signing key and package name. The installation process assigns a locally unique Linux user identifier (UID) to each installed application and at runtime Android applications can be identified either based on UID or package name (one should note that package names are not globally unique, developers may freely use any package name). Only applications that are signed with the same key can be assigned the same UID.

MSSF. During installation applications are identified by software source (signing key) and package name. At runtime applications can be identified by a globally unique application identifier that consists of three parts: software source, package name and package-specific application identifier.

6.2 Application update

Symbian. Centrally signed Symbian application can only by updated by an installation package that has been assigned the same SID by the central trusted authority. Self-signed applications can be updated by any installation package that has the same SID from the unprotected range.

Java ME. Update of signed midlets is allowed only if the new midlet package is signed by the same key as the previous version was. This approach is often called *same-origin policy*. For signed midlet update the application persistent storage is retained. In unsigned application update the user is asked whether the new application should have access to the persistent storage of the old application.

Android. In Android, application update is always based on same-origin policy, i.e. an application can be updated only from an installation package that is signed with the same developer key as the currently installed application. Updated application gets access to the same data storages as its previous version.

MSSF. Application can be updated if the installation package is signed by the same software source from which it was originally installed or by another software source that has higher trust level. Also in MSSF, the updated application automatically gets access to data of previous application version.

6.3 Permission granularity

Symbian. Symbian platform security architecture defines a fixed number (21) of permissions (capabilities). The capabilities are divided into four categories: User Capabilities can be granted by the user during application installation. System Capabilities require application signing by the central trusted authority. Restricted Capabilities require application signing with stronger application developer identity checking (publisher identity certificate). Manufacturer Capabilities are reserved for device manufacturer.

Java ME. Java ME provides more fine-grained permission set. System API developers can define their own permissions. The permissions are mapped into coarse-grained "function groups". The purpose of the function groups is to present permission requests to the user in human understandable format. The MIDP specification recommends 15 function groups for mobile devices.

Android. The default permission set defined by Google includes 112 permissions. The permission names are intended to be user understandable, but in practice this is not the case with all permissions (e.g. a permission named "BROADCAST_STICKY"). Android developers may define their own permissions, and thus the number of permissions is unlimited. Permissions are categorized into four protection levels: Normal, Dangerous, Signature and SystemOrSignature.

MSSF. In MSSF architecture access control can be defined using traditional Linux access control mechanisms and using permissions (resource tokens). Resource tokens names should be understandable for users. MSSF provides a standard set of global resource tokens. Additionally, applications can define their own local resource tokens. The number of permissions in MSSF is unlimited.

6.4 Permission assignment

Symbian. A developer declares the requested permissions in MMP file. For most applications the permission assignment is done by the central trusted authority during application signing. During application installation the Symbian installer validates application signature if System,

	Symbian	Java ME	Android	MSSF
application identification at runtime	application and vendor identifier assigned by central authority	not applicable	local UID and package name	sotfware source, package name and package-specific application identifier
application update	application identifier assigned by central authority	same-origin policy and user approval	same-origin policy	trust level of software source
permission granularity	coarse-grained	fine-grained and coarse-grained groups	unlimited permissions	unlimited permissions
permission assignment	signature by central authority and user at installation	signature by protection domain owner and user at runtime	user at installation	signature by software source
runtime application integrity	dedicated directory (manufacturer permissions)	Java sandboxing	Linux access control and Java sandboxing	Linux access control and permissions and IMA [27]
offline application integrity	not supported	not supported	not supported	hardware-assisted EVM [31]
access control policy declaration	permissions and application and vendor identifier	permissions	permissions	permissions and Linux access control and application identifier
access control policy scope	system APIs and application IPC	system APIs	system APIs and application IPC	system APIs and application IPC and file access
runtime application data protection	dedicated directory (manufacturer permissions)	Java sandboxing	dedicated directory	fine-grained permission-based policies
offline application data protection	hardware-assisted with restricted API	not supported	not supported	hardware-assisted with file system integration

Table 1: Summary of key features in mobile platform security architectures.

Restricted or Manufacturer Capabilities are requested. For requested User Capabilities a user prompt is generated during application installation.

The permission set that an application gets during installation remains the same throughout the application lifetime. When an executable loads a library, the operating system checks the capabilities of the library. The library must have all capabilities of the executable, otherwise loading fails. (Because of this Symbian system libraries typically have almost all capabilities.)

Java ME. The developer declares the permissions that his application needs in JAD or manifest file. When an application is installed, the signature is checked against the protection domains on the device. If the protection domain does not support the requested permissions the installation is denied. At application runtime API calls that require user-grantable permission trigger a prompt to the user. The prompts are presented in terms of function groups. If the user grants the requested permission, the decision applies to all permissions of the same function group. The user can grant permissions to a midlets permanently, for midlet execution lifetime or for one-time access. The permissions of a midlet remain constant during midlet lifetime.

Android. Android developers declare the requested permissions in a manifest file. Normal permissions do not require explicit user granting. Each application that requests such permissions will get them. Dangerous permissions must be granted by the user during application installation. Signature permission can be given to an application, only if it is signed by the same key as the application that declared the permission. SystemOrSignature permissions are additionally granted to OS manufacturer applications. The permissions set that an Android application gets during application installation remains constant during application lifetime.

MSSF. Developer declares requested permissions (resource tokens) in the application manifest file. The manifest file can additionally declare a requested Linux UID, GID and POSIX permissions. When an application is installed, the installer checks the requested permissions against the permissions of the software source from a local policy file. The application is granted the intersection of these two permission sets. A special resource token type, unique application identifier, is generated by the installer for all applications.

At runtime applications can request the kernel to drop some of their permissions. When an application loads a shared library, its set of permissions stays the same, but the library loading may fail if a library comes from a software source that cannot grant all the permissions that the application currently possesses. The permission set of an installed applications can also increase during the lifetime of the application. This happens when a plugin library is installed as an extension to an already installed application.

If the plugin requires permissions currently not possessed by the application, these permissions can be added to the permission set of the application, if software sources of both the application and the plugin are allowed to grant the missing permissions.

6.5 Application integrity

We use term "application integrity" to refer to the protection of installed applications against unauthorized modifications both when the system is running (runtime application integrity) and when the device is powered down (offline application integrity).

Symbian. Executable files are kept in and exclusively loaded from a dedicated directory (/sys/bin) in the device internal memory. Only processes with manufacturer capabilities are allowed to access this directory which provides runtime application integrity.

Symbian supports also application installation to removable memory elements. In such a case, the installer calculates a hash of the executable binary and stores the hash to device internal memory and the executable itself to removable element. When an executable is loaded from the removable memory element, a hash of the executable is calculated again and compared to the one stored on internal memory.

Symbian platform security model does not support offline application integrity protection. Instead, the platform security architecture relies on the assumption that the device internal memory cannot be accessed, and thus the already installed applications cannot be modified, by the attacker when the device is powered down.

Java ME. Midlets are executed in a sandbox of Java virtual machine and do not have direct access to the device file system which prevents them from modifying each other. Java ME platform security architecture does not address offline application integrity.

Android. Applications are stored in directories that are assigned unique Linux UIDs during installation. Standard Linux access control mechanisms prevent applications from modifying each other. Additionally, Android devices do not have root account available and third party applications cannot run with root UID, which preserves integrity of these directories. Java sandboxing prevents applications from modifying their own stored attributes such as permissions.

MSSF. Running processes as root is not explicitly prevented in MeeGo devices, and thus relying on UID-based access control is not enough. Instead a combination of Integrity Measurement Architecture (IMA) [27] and Extended Validation Module (EVM) [31] is used. During application installation a reference hash for executable binaries (and also other executable file, such as scripts) is calculated. The reference hashes are stored in an extended Linux file system attribute called security.ima for each file. IMA/EVM verifies these hashes when applications are executed. The security.ima attribute is automatically recalculated, when an application binary is modified during system run-time. MSSF uses the Smack kernel module in order to enforce the access control permissions of the filesystem in addition to standard Linux filesystem permissions.

The offline integrity of security.ima attribute and other file attributes is preserved using hardware-based TrEE. EVM module calculates a keyed message authentication code using a key that is protected by TrEE. This prevents unnoticed modification of security.ima when the device is powered down.

6.6 Access control policy

We use term "access control policy" to refer to both the declaration and scope of rules that control access to the system APIs and IPC services provided by installed applications.

Symbian. Symbian servers can provide services to other Symbian executables through Symbian IPC framework. Developer of a Symbian server defines access control policy for the server APIs by assigning required permissions and application identifiers (SID and VID) for each API function call. The access control policy declaration is done by writing C++ code. The Symbian IPC framework automatically enforces permission-based access control rules for IPC function calls (SID or VID based access control enforcement must be implemented in code).

Java ME. In Java ME platform system APIs provide access to protected resources. Midlets themselves cannot communicate with each other in current MIDP version. System API developers define access control policies by assigning required permission to appropriate API function calls in Java code. The API developer may either reuse existing permissions or define their own.

Android. Android applications can provide services to each other through Android IPC framework. Developers declare access control policies by defining the set of permissions that are needed to use entire service (defined in manifest file) or to use an individual function from an IPC service interface (implemented in code). Also Android IPC framework automatically enforcement permission based access control rules.

MSSF. Applications can communicate with each other through D-Bus and local socket interfaces. Application developers may declare access control policies for such IPC in terms of permissions (resource tokens) or traditional Linux access control mechanisms (e.g. UID or GID). The access control policy declaration is done in the application manifest file. Developers can use common system wide permissions or declare their own application specific permissions. Unlike in other platforms MSSF access control policies can also be defined for file access (e.g. an application developer may define the resource tokens needed to access any of the files created by the application). Internally, the access control enforcement is done by Smack kernel module [29].

6.7 Application data protection

We use term "application data protection" to refer to the protection of application persistent storage (e.g., files on device file system) against unauthorized modification and eavesdropping both when the device is running (runtime protection) and powered down (offline protection).

Symbian. In Symbian platform security model a dedicated directory is created in the file system for each application. This directory can only be accessed by the owner application or a process with manufacturer capabilities. Application developers may define whether the contents of application private directory should be included to backups that are made from the device data. The default policy is to exclude private directory contents from backups.

Nokia Symbian devices provide a restricted API for sealing (authenticated encryption) data with TrEE-resident device

key (or derivation of it). This feature provides support for offline data protection for certain Symbian applications.

Java ME. In Java ME architecture applications do not have direct access to device file system, instead database system is provided for persistent storage. Databases are either private to the application itself or shared with all other applications on the same device. Java ME platform security model does not address offline application data protection.

Android. Each application has a dedicated directory protected with Linux UID and GID. By default, the files in this directory can be written and read only by the application itself. During file creation, the application may explicitly define that the created file should be readable or writable by other applications as well. Android platform provides an automatic backup feature that creates backups of application data to an online server. These backups are protected by Google account authentication.

MSSF. MSSF architecture provides fine-grained data caging model. Application developers can define in manifest file required permissions for each type of file access (read, write etc.) for each application file. By default applications files can be accessed only by the application itself.

MSSF allows applications to encrypt data using a TrEE-resident device key. The key derivation can be based on application identifier or specified resource token which allows an application to encrypt data only for itself or to a set of applications. This encryption feature is integrated to the device file system which allows legacy applications to utilize offline data protection without changes.

7. DISCUSSION

Modern mobile platform security architectures have generously borrowed from old ideas but with new twists to adapt these ideas to the needs of mobile devices. For example, all of the software security architectures discussed in Section 4 incorporate access control schemes built around the notion of permissions that are granted to the subjects and checked at the time of access control. This is similar to the VAX/VMS notion of "privileges" introduced back in the late 1970s [24]. However, while privileges in VAX/VMS are typically granted to a user of the system, in modern platform security architectures, they are granted to software modules. Similarly, the notion of secure bootstrapping was discussed in the 1990s [2], but saw widespread adaption when smartphones started to be deployed. As described in Section 5 MSSF borrows from the notion of "authenticated boot" introduced by the Trusted Computing Group, with a new twist: allowing any OS software image to be booted, but if the booted image is not authorized by a trusted party (e.g., device manufacturer), the user is alerted to the fact; furthermore access to sensitive resources (such as cellular network access) or data (such as encrypted device-specific keys) are rendered inaccessible by the OS.

Other examples of old techniques adapted, sometimes with new twists, include the use of code signing for code identification and as the basis for permission assignment. Offline data caging (discussed in Section 6.7) makes use of hardware-assisted secure storage similar to the notion of sealing in Trusted Computing Group specifications, but with the addition that sealed data can be bound to application identities or permissions.

Despite the widespread deployment of mobile platform security architectures, a number of open problems remain.

7.1 Permission granularity

Symbian platform security took the approach of using a fixed number of pre-defined permissions. Resource and service providers who need access control attempt to find the most relevant pre-defined permission to protect their resource or service but may not always succeed. This may lead to developer confusion.

Java ME, Android and MSSF on the other hand allow fine-grained permissions and make it possible to define new permissions. But this does not completely address the problem either. Having numerous permissions can be a cause of potential confusion among users and developers [5]. For this reason some of these architectures allow permissions to be grouped together for presentation to the user.

Designing a flexible and sufficiently rich platform security architecture without sacrificing usability remains a challenge. Solutions like Security Enhanced Linux are not widely adapted because of their perceived complexity. This choice may need to be revisited.

7.2 Permission assignment

Perhaps the most serious problem with mobile platform security architectures is the issue of permission assignment. Android (and also partially Java ME and Symbian) take the approach of relying on the user to decide whether an application can be grated dangerous permissions. Two basic approaches are used: the user can be prompted to grant permissions during application installation (Android and Symbian) or during application runtime (Java ME). Both of these approaches have drawbacks. Runtime permission prompts are often considered annoying by the users while permission assignment during application installation suffers from the lack of relevant context (e.g., the user might know only at runtime whether Internet access should be granted to an application). In both cases users are ill-equipped to make permission assignment decisions and may become habituated to click-through access control prompts.

7.3 Software appropriateness

The problems of user-based permission assignment can be avoided by having a central authority that does permission assignment by means of code signing as in the case of iTunes AppStore or SymbianSigned. While centralized permission assignment is preferable from a usability perspective, it is problematic in cases where subjective judgement is involved. This is particularly so in cases where centralized judgement regarding the appropriateness of applications (e.g., classifying applications as offensive or otherwise inappropriate) is made [8]. One alternative is to rely on the wisdom of crowds (e.g., the WhatApp service at `http://whatapp.org`) or the wisdom of smaller and more personalized "cliques" [8, 7].

7.4 Access control policy enforcement

Two basic approaches for platform internal access control policy enforcement exist. First, a *reference monitor* (e.g. the platform security architecture on a mobile device) can enforce access control policies over *subjects* (e.g. caller application and callee application in IPC communication) [1]. Second, the full responsibility of access control enforcement can be left to the callee application, assuming that the underlying platform provides the needed information about the caller to the callee.

Most of the platform security architectures described in

this paper use a hybrid approach. The platform security architecture typically enforces access control policies defined in application manifest files automatically, but the application developers can implement additional access control checks, e.g. based on IPC call input data, on top. In Symbian and MSSF callee application can query attributes of caller, such as application identity and possessed permissions.

Pure reference monitor approach preserves application privacy, i.e. the callee does not learn more about the caller than what is needed to make the access control decision, while providing full information about the caller to the callee enables useful security services, such as attestation of untrusted application attributes to an external party by a trusted callee application. Finding the optimal balance between these two approaches remains an open challenge.

7.5 Colluding applications

All of the mobile platform security architectures discussed above grant and enforce accesss control to individual pieces of software. Two such pieces could collude over overt or covert inter-process communication channels so that they gain access to resources or services that neither was able to acting along [18]. Defending against this problem appears to be very hard. In particular, in cases where the user is responsible for granting permissions, visualizing the different potential collusion scenarios and their implications to the user is a security usability challenge.

8. CONCLUSIONS

"Smartphone security" is becoming a popular research topic. The fundamental security issues with smartphones are also present in a larger class of personal mobile communication devices, as well as in personal computers. However, unlike PC platforms, all dominant mobile platforms incorporate widely deployed mobile platform security architectures. The history of mobile platform security goes back long beyond the current popularity of smartphones. The widespread deployment of mobile platform security architectures is due to specified and perceived business, regulatory and end-user requirements for mobile communication devices.

We surveyed four mobile platform security architectures. In all of these architectures the fundamental concepts are borrowed from older commercial or research systems, but some of them have been adapted with new twists to suit the needs of mobile platforms. We also discussed a number of issues that are insufficiently addressed by the current generation of mobile platform security architectures. They constitute fertile ground for further research on the topic.

9. REFERENCES

[1] James Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, Electronic Systems Division, 1972.

[2] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. A secure and reliable bootstrap architecture. In *IEEE Symposium on Security and Privacy*, pages 65–71. IEEE Computer Society, 1997.

[3] ARM. Trustzone-enabled processor. `http://www.arm.com/products/processors/technologies/trustzone.php`.

[4] ARM. *Building a Secure System using TrustZone^{TM} Technology*, 2009. Available from `http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf`.

[5] David Barrera, Hilmi Günes Kayacik, Paul C. van Oorschot, and Anil Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *ACM Conference on Computer and Communications Security*, pages 73–84. ACM, 2010.

[6] Desktop bus project page. website. `http://www.freedesktop.org/wiki/Software/dbus`, 2010.

[7] Pern Hui Chia, Andreas Heiner, and N. Asokan. Use of ratings from personalized community for trustworthy application installation. In *Proceedings of the the 15th Nordic Conference in Secure IT Systems*, 2010.

[8] Pern Hui Chia, Andreas Heiner, and N. Asokan. The wisdom of cliques: Use of personalized social rating for trustworthy application installation. Technical Report NRC-TR-2010-001, Nokia Research Center, July 2010. Available at `http://research.nokia.com/files/tr/NRCTR2010001.pdf`.

[9] Jan-Erik Ekberg and Markku Kylänpää. Mobile trusted module. Technical Report NRC-TR-2007-015, Nokia Research Center, November 2007. Available at: `http://research.nokia.com/files/NRCTR2007015.pdf`.

[10] ETSI. *ETSI GSM 02.09 Security Aspects*. European Telecommunication Standards Institute, April 1993. Version 3.1.0; Available from `http://www.3gpp.org/ftp/Specs/html-info/0209.htm`.

[11] ETSI. *ETSI GSM 02.09 Security Aspects*. European Telecommunication Standards Institute, June 2001. Version 8.0.1 Release 99; Available from `http://www.3gpp.org/ftp/Specs/html-info/0209.htm`.

[12] Gartner. Press release; worldwide mobile phone sales in trhid quarter 2010. `http://www.gartner.com/it/page.jsp?id=1466313`, 2010.

[13] Gitorious. Mssf project source code. `http://meego.gitorious.org/meego-platform-security`, 2010.

[14] Dmitry Kasatkin. Mobile simplified security framework. In *Proceedings of the 12th Linux Symposium*, 2010.

[15] Butler Lampson. Protection. In *Proceedings of the 5th Princeton Conference on Information Sciences and System*, pages 18–24, 1971.

[16] Steve Litchfield. Defining the smartphone. On-line article at AllAboutSymbian.com, July 2010. Available at `http://www.allaboutsymbian.com/features/item/Defining_the_Smartphone.php`.

[17] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the linux operating system. In Clem Cole, editor, *USENIX Annual Technical Conference, FREENIX Track*, pages 29–42. USENIX, 2001.

[18] Claudio Marforio, Srdjan Capkun and Aurélien Francillon. Personal communication, November 2010. Paper in submission.

[19] Jonathan M. McCune, Bryan Parno, Adrian Perrig,

Michael K. Reiter, and Arvind Seshadri. Minimal TCB Code Execution (Extended Abstract). In *Proc. IEEE Symposium on Security and Privacy*, May 2007.

[20] MeeGo. Mobile simplified security framework overview. `http://conference2010.meego.com/session/mobile-simplified-security-framework-overview`, 2010.

[21] Sun Microsystems. Mobile information device profile for java 2 micro edition, version 2.1. `http://www.oracle.com/technetwork/java/index-jsp-138820.html`, 2006.

[22] Motorola. Mobile information device profile for java micro edition, version 3.0. `http://opensource.motorola.com/sf/projects/jsr271`, 2009.

[23] Oracle. Java technology. `http://www.java.com/en/about/`, 2010.

[24] Hewlett Packard. Openvms guide to system security. Available from `http://www.hp.com/go/openvms/doc/`, June 2010.

[25] Siani Pearson, editor. *Trusted Computing Platforms: TCPA technology in context*. Prentice Hall, 2003.

[26] Elena Reshetova. Mobile simplified security framework overview. `http://userweb.kernel.org/~jmorris/lss2010_slides/reshetova_LinuxCon_overview_v_final.pdf`, 2010.

[27] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and implementation of a tcg-based integrity measurement architecture. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, pages 16–16, Berkeley, CA, USA, 2004. USENIX Association.

[28] Jane Sales. *Symbian OS Internals*. Wiley, 2005.

[29] Casey Schaufler. Smack in embedded computing. In *Proceedings of the 10th Linux Symposium*, 2008.

[30] Dries Schellekens, Pim Tuyls, and Bart Preneel. Embedded trusted computing with authenticated non-volatile memory. In *Proc. of the 1st International conference on Trusted Computing and Trust in Information Technologies (TRUST 2008)*, 2008.

[31] SourceForge. An overview of the linux integrity subsystem. `http://heanet.dl.sourceforge.net/project/linux-ima/linux-ima/Integrity_overview.pdf`, 2010.

[32] Jay Srage and Jerome Azema. M-Shield mobile security technology, 2005. TI White paper. `http://focus.ti.com/pdfs/wtbu/ti_mshield_whitepaper.pdf`.

[33] Harini Sundaresan. OMAP platform security features, July 2003. TI White paper. `http://focus.ti.com/pdfs/vf/wireless/platformsecuritywp.pdf`.

[34] Trusted Computing Group. `https://www.trustedcomputinggroup.org/home`.

[35] TCG. Trusted Platform Module (TPM) Specifications. Available at: `https://www.trustedcomputinggroup.org/specs/TPM/`.

[36] Android-DLS wiki. Howto: Unpack, edit, and re-pack boot images. `http://android-dls.com/wiki/index.php?title=HOWTO:_Unpack%2C_Edit%2C_and_Re-Pack_Boot_Images`, 2010.

[37] Maurice Wilkes. *The Cambridge CAP computer and its operating system*. North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, 1979.

Publication II

Reshetova, Elena and Bonazzi, Filippo and Nyman, Thomas and Borgaonkar, Ravishankar and Asokan, N. Characterizing SEAndroid Policies in the Wild. In *2nd International Conference on Information Systems Security and Privacy*, Location, pages, and other detailed information, October 2016.

Characterizing SEAndroid Policies in the Wild

Keywords: Security, Access Control, SELinux, SEAndroid

Abstract: Starting from the 5.0 Lollipop release all Android processes must be run inside confined SEAndroid access control domains. As a result, Android device manufacturers were compelled to develop SEAndroid expertise in order to create policies for their device-specific components. In this paper we analyse SEAndroid policies from a number of 5.0 Lollipop devices on the market, and identify patterns of common problems we found. We also suggest some practical tools that can improve policy design and analysis. We implemented the fist of such tools, SEAL.

1 INTRODUCTION

During the past decade Android has gained a considerable share of the mobile device market. However, at the same time the number of malware and various exploits available for Android has also been increasing (Zhou and Jiang, 2012; Smalley and Craig, 2013). Many classical Android exploits, such as GingerBreak and Exploid (Smalley and Craig, 2013), attempt to target system daemons that run with elevated, often unlimited, privileges. Once such a daemon is compromised, the whole Android OS usually becomes compromised and the attacker is able to get permanent root privileges on the device. Since the Android permission system, which relies on Linux Discretionary Access Control (DAC), cannot protect from such attacks, a new Mandatory Access Control (MAC) mechanism has been introduced. SEAndroid (Smalley and Craig, 2013) is an Android port of the well-established SELinux MAC mechanism (Smalley et al., 2001) with some Android-specific additions and modifications. In SELinux, security decisions are taken according to a policy: the reference policy for SEAndroid was created from scratch and is maintained as part of the Android Open Source Project (AOSP) [1].

Starting from the Android 5.0 Lollipop release, the Android compliance requirements have mandated that every process must be run inside a confined SEAndroid domain with a proper set of access control rules defined. This has put many Android Original Equipment Manufacturers (OEMs) in the difficult position of enabling SEAndroid in enforcing mode on their devices with a set of fully configured access control domains. While the reference SEAndroid policy is provided by AOSP, any OEM customization to the reference AOSP device design results in a need for SEAndroid policy modifications. Writing well-designed SELinux policies requires expertise; this difficulty, together with high time-to-market pressure, can possibly lead to the introduction of mistakes and outright vulnerabilities in modified SEAndroid policies deployed in OEM Android devices.

In this paper, we conduct a systematic manual analysis of several available SEAndroid 5.0 Lollipop OEM policies and identify common patterns and mistakes. We find that OEM modifications can render policies less strict, resulting in a wider attack surface for potential vulnerabilities. Based on these findings, we identify a number of practical tools that can assist SEAndroid policy designers and researchers to analyze and improve SEAndroid policies. We also provide an initial implementation of one such tool, *SEAL*. To the best of our knowledge, this is the *first comparative study* of SEAndroid policies from real-world devices.

2 BACKGROUND

2.1 SELinux

SELinux (Smalley et al., 2001) is a well-established MAC mechanism available for Linux-based distributions. It was the first MAC for mainstream Linux with its initial release in 1998. SELinux has been implemented in the Linux kernel following the Flask architecture (Spencer et al., 1999), where the policy enforcement code, known as Linux Security Module (LSM) Framework, and the policy decision-making code are separated: this allows other MAC modules, such as AppArmor (Bauer, 2006) or Smack (Schau-

[1] source.android.com

fler, 2008), to utilize the same policy enforcement code.

The main MAC mechanism in SELinux is Domain/Type Enforcement (Badger et al., 1995), which assigns a `type` to each subject or object in the system; a subject's type is also known as `domain`. A subject running in `domain` can only access an object belonging to `type` if there is an `allow` rule in the policy of the following form:

```
allow domain type : class permissions
```

where `class` represents the nature of an object such as file, socket or property, and `permissions` represent the types of operation on this object that are being controlled, like `open`, `write`, `set` etc. Subjects can change their `domain` if a corresponding type transition rule is defined. For example, if a process executes a new binary, it is possible for the resulting process to run in a different `domain`. Such a transition rule will be represented as:

```
type_transition olddomain type:process
                newdomain
```

where `type` denotes the type of a binary that should be executed in order for the transition to happen. We will use the term *process transition* in the future to refer to such a rule. In practice there are a number of additional rules that are needed in order to make the transition happen, but we leave them out here for the sake of simplicity. Another type of transition can occur if an object is created and its `type` should differ from the `type` of the object's parent. For file creation inside an existing directory such a rule can be represented as:

```
type_transition domain oldtype:dir newtype
```

where `domain` denotes the domain of the subject creating the file, `oldtype` represent the `type` of the directory where the file is being created and `newtype` denotes the `type` that the new file should be assigned.

In addition, the SELinux policy language has the following notions:

- An `attribute` is a way to refer to sets of types and domains. It is used to express type hierarchies and rule inheritance. For example, SELinux policies on Android define an `app` attribute consisting of common rules for all platform applications such as the ability access the device display.
- *Initial* `Security Identifier (SID)` are types that should be assigned by default to subjects and objects during system initialization, such as for example `kernel` and `init`.
- `genfs` contexts define types that should be assigned to objects residing in special filesystems, such as `proc`, `debugfs` and `ecryptfs`.

2.2 SEAndroid

The SELinux port to Android, SEAndroid (Smalley and Craig, 2013), was mostly based on SELinux code with some additional LSM hooks to support Android-specific mechanisms, such as Binder Inter Process Communication (IPC). However, the SEAndroid reference policy was written from scratch due to 1) a desire for a simpler and a smaller policy and 2) the big difference between the userspace layers of Android and a standard Linux distribution.

SEAndroid classes and permissions are mostly the same as on SELinux, with some Android-specific additions like the `property` class for the Android init-based property service and the `keystore_key` class for the Android keystore key object.

Native services and daemons are assigned SEAndroid domains based on filesystem labeling or direct domain declaration in the service definition in the `init.rc` file. In turn, applications are assigned domains based on the signature of the Android application package file (`.apk`). There are a number of predefined application domains, like `system_app`, `platform_app` and `untrusted_app`. OEMs are able to create additional domains if needed.

One new notion that SEAndroid has is the presence of `neverallow` rules in the source policy. A `neverallow` rule specifies that certain accesses should never be allowed by the policy. For example, the following `neverallow` rule asserts that only processes running in the `init` domain should be able to modify security-sensitive files in the `proc` filesystem:

```
neverallow {domain -init}
proc_security:file {append write}
```

If one tries to add a rule that conflicts with a `neverallow` rule, the policy compilation fails.

SEAndroid was initially added to the AOSP codebase for Android 4.3 back in 2012; at that point, it was configured in permissive mode. In Android 4.4, SEAndroid was switched to enforcing mode: however, most domains were left in permissive mode, apart from a number of core AOSP domains such as `init` and `vold`. Android 5.0 Lollipop eventually required every single process to be put in an enforcing domain, effectively extending the enforcement to the whole system.

2.3 OEM Modifications to the AOSP SEAndroid Reference Policy

Default AOSP services, processes and applications are already covered by the AOSP SEAndroid reference policy. Typically, OEM Android devices are

highly customized with their own specific drivers, new services, processes and filesystem mounts. In order for these custom components to work, appropriate additions must be made to the SEAndroid reference policy. OEMs were allowed to make additions to the SEAndroid reference policy right from the start, but very few of them actually did in Android 4.3 and 4.4: and in fact, the resulting policy was stricter than the AOSP reference policy. The stringent requirements of Android 5.0 Lollipop, however, forced all OEMs to deploy comprehensive policies defining complete rules for their own custom services: this turned out to be a challenging task for most. The inherent difficulty of incorporating SEAndroid in the development process, combined with high time-to-market pressure, has resulted in the introduction of anti-patterns, mistakes and potential vulnerabilities in OEM policies.

3 SEANDROID POLICIES IN DEPLOYED OEM DEVICES

3.1 Statistical Analysis

We collected 8 policy files from non-rooted, off-the-shelf commercial Android 5.0 Lollipop devices by different manufacturers. Table 1 shows the comparison of all basic policy attributes and characteristics with regards to the Android 5.0 Lollipop AOSP SEAndroid reference policy in the following categories:

- **Policy size.** All OEMs increased the policy size, by factors ranging from 1.1 up to 3.2.
- **Types, domains, type transitions and domain transitions.** The overall ratio of newly added domains to newly added types ranges between 4.7 and 6.5, which is very close to the ratio in AOSP itself (6.3). We conjecture that OEMs tend to add slightly simpler domains, with fewer types per each domain defined. The ratio of newly added process transitions to newly added domains is close to 1; this indicates that OEMs add simple domains, with only one process transition to these domains either from the `init` domain (upon system startup) or from the parent process domain (usually upon execution of processes such as `shell` or `toolbox`). This ratio is 1.4 for LG G3, due to a number of newly defined domains and having two or more transitions to the `shell`, `toolbox`, `dumpstate` and `logcat` domains. New type transitions are mostly used by OEMs for `tmpfs` types, as in the following example:

```
type_transition aal tmpfs : file
                aal_tmpfs
```

- **Allow rules.** The ratio of total number of allow rules to total number of types varies between 10.9 and 13.1, with the exception of 14.8 for Motorola G and 18.7 for LG G3; the ratio for AOSP is 12.0. The numbers for Motorola and LG are comparatively excessive, and may indicate overly permissive policies; this may be due to the use of tools to automatically generate policies from system logs.
- **Attributes.** Only Samsung and Sony define new attributes. Sony adds only one, probably related to the system update process. In contrast, Samsung adds many new attributes, which seem to be auto-generated and most probably used for policy optimization. The rest of the OEMs add separate domains for their services and applications, and do not introduce any new domain hierarchies: this may imply unfamiliarity with the use of policy hierarchies.
- **Classes, permissions and initial SIDs.** OEMs do not modify the default set of SEAndroid `classes` (86), `permissions` (267) or initial `SIDs` (27): this is to be expected, since they represent interfaces and objects recognized and supported by SEAndroid. The only change we observed was for Samsung S6, that had 4 more permissions defined: `delete_as_user`, `get_by_uid`, `insert_as_user` and `set_max_retry_count`, all granted on the `keystore_key` class. The most probable reason for such additions is Samsung's implementation of the keystore and its API, which requires specific permissions.
- `genfs` **contexts.** The primary reason for the addition of `genfs` contexts is that most OEMs have additional mount points and filesystems on their devices, which by default would be labeled as `unlabeled` unless a proper `genfs` context for it is specified. An AOSP `neverallow` rule prohibits any OEM domain from creating files with this type: this restriction has forced OEMs to define proper types for their new mount points.

3.2 Systematic Manual Analysis

We manually searched each policy for OEM misconfigurations: we used existing tools for SELinux policy analysis, which we found to be cumbersome. Our primary tool was `apol` (Tresys, 2014), a GUI tool that allows the user to load a binary policy and examine it by specifying various filters. However, `apol` was not suitable for comparing two policies: it was necessary to run two instances of `apol` simultaneously,

Table 1: Policy comparison and complexity.

	size (KB)	*types*	*domains*	*type trans*	*process trans*	*allow rules*	*attributes*	*genfs contexts*	*untrusted_app rules*
AOSP	117	341	54	95	41	4096	21	30	33
LG Nexus 5	134	416, +75	65, +11	158, +63	51, +10	4972, +876	21	32, +2	33
Intel	127	393, +52	65, +11	115, +20	51, +10	4748, +652	21	32, +2	38, +5
HTC M7	181	621, +280	106, +52	213, +118	95, +54	7587, +3491	21	34, +4	46, +13
Motorola G	193	590, +249	92, +38	199, +104	83, +42	8753, +4657	21	33, +3	33
LG G3	302	851, +510	149, +95	340, +245	180, +139	15921, +11825	21	45, +15	168, +135
Intex Aqua	230	900, +559	142, +88	266, +171	128, +87	9824, +5728	21	37, +7	44, +11
Samsung S6	370	1102, +761	215, +161	430, +335	180, +139	14412, +10316	158, +137	43, +13	81, +48
Sony Xperia	218	793, +452	139, +85	265, +170	113, +72	9308, +5212	22, +1	37, +7	42, +9

+ denotes the number of additions compared to AOSP

manually insert the same queries into both and examine the differences between the outputs. Another tool was `sediff` (Tresys, 2014), which can do basic policy comparison but does not allow filtering based on specific types or domains.

To make our manual analysis tractable, we identified three sets of types that we consider important to check. The first set comprises core Android and security-sensitive domains, such as `init`, `vold`, `keystore`, `tee`, as well as types that protect access to security-sensitive areas of the filesystem, such as `proc_security`, `kmem_device` and `security_file`. The second is the set of default types that would be assigned to an object upon its creation unless a concrete type is specified in one of the policy files. The third is the set of types that would be assigned to untrusted code and its data, primarily the `untrusted_app` domain.

Analyzing these sets of types and the associated rules, we discovered the following patterns across many devices from different OEMs.

3.2.1 Overuse of Default Types

As mentioned above, a default type is one that is assigned to an object upon creation unless a dedicated type for it is specified in the policy files: examples include `unlabeled`, `device`, `socket_device`, `default_prop` and `system_data_file`. Table 2 shows that in many cases OEMs overuse the default SEAndroid object types. For example, compared to AOSP, HTC M7 has 10 new rules allowing various system daemons, such as `healthd`, `netd`, `vold`, `mediaserver`, `wpa`, `system_server`, to set system properties with the default type `default_prop`. In practice, this means that some of the system properties belonging to these components end up labeled as `default_prop`. Similarly, HTC M7 has 13 more rules granting various system daemons (`rild`, `mediaserver`, `thermal-engine`, `sensors`, `thermald`, `system-server`, ...) `write` access to the default `socket_device` object type. The only exceptions when OEM actually reduced the number of rules with regards to default type `device` is LG Nexus 5 and Motorola G policies, where they removed a rule belonging to `logd`. Below are concrete example rules from different OEMs to show the usage of default types:

```
allow thermald socket_device : sock_file
    {write create setattr unlink}

allow mediaserver default_prop :
    property_service set

allow untrusted_app unlabeled : dir
    {ioctl read getattr search open}

allow untrusted_app unlabeled :
    filesystem getattr
```

Plausible reasons for OEMs to use default types include the fact that objects are automatically assigned default types, and the common practice of using tools like `audit2allow` (SELinux Project, 2014) which parse audit logs and automatically create new allow rules to permit denied accesses.

There are two main consequences of such mistakes. Foregoing distinct, dedicated types in favor of default types means that different, unrelated resources are collected under a common label: domains with access to said label thus get wider access rights than actually needed. This is undesirable, as it violates the principle of least privilege. The second, more severe, consequence is that some untrusted domains might be given access to default labeled sensitive objects.

Fortunately, we did not find examples of such cases in the policies we examined, apart from the example above where `untrusted_app` is given some access to `unlabeled` filesystem objects; however, the possibility of such mistakes remains.

Google is actively trying to address this problem by fine-tuning the set of `neverallow` rules in the AOSP reference policy. Starting from Android 5.1, OEMs are not allowed to modify this set. For example, it is not possible anymore for an OEM domain to set default properties or access block devices.

Table 2: Usage of default types by OEMs.

	unlabeled	***socket_device***	***device***	***default_prop***	***system_data_file***
AOSP	25	4	18	0	42
LG Nexus 5	25	7, +3	17, –1	0	45, +3
Intel	27, +2	5, +1	24, +6	3, +3	54, +12
HTC M7	31, +6	17, +13	26, +8	10, +10	68, +26
Motorola G	31, +6	7, +3	17, –1	2, +2	62, +20
LG G3	42, +17	21, +17	28, +10	3, +3	120, +78
Intex Aqua	33, +8	8, +14	23, +5	0	108, +66
Samsung S6	24, –1	22, +18	46, +28	1, +1	204, +162
Sony Xperia	25	15, +11	21, +3	1, +1	57, +15

± *denotes the number of additions/removals compared to AOSP*

3.2.2 Overuse of Predefined Domains

Another observed trend is that typically OEMs do not define separate domains for specific system applications, but tend to place them either in `system_app` or in `platform_app` domains. Consequently, these domains accumulate a lot of `allow` rules that are shared by all system or platform applications. Moreover, if many applications are pre-installed in the same domain, SEAndroid cannot prevent privilege escalation attacks or unauthorized data sharing by such apps (Smalley and Craig, 2013). As an example, let us consider the pre-installed McAfee anti-virus application on LG devices. It runs in the `system_app` domain, which contains more than 900 associated allow rules in the LG G3 policy compared to 46 in the AOSP one. It is quite difficult to identify specific rules that were added to the `system_app` domain because of the McAfee application, given that many other applications run in this domain. However, by analyzing the permissions of the same McAfee application in Google Play Store, we observed corresponding SEAndroid policy rules in the `system_app` domain, such as access to the telephony functionality, camera and several types related to the filesystem, including `tmpfs` types and sockets. This might indicate that these rules were added for the McAfee application.

A solution to this problem would be to place certain powerful system applications in their own SEAndroid domains; this can be done by signing these applications with different keys and creating a mapping between these keys and target application domains.

3.2.3 Forgotten or Seemingly Useless Rules

Another common trend is the presence of rules that seem to have no effect. One example is rules of the following type present in one device:

```
allow untrusted_app <xyz>_exec : file
        <file op>
```

For example

```
allow untrusted_app tee_exec : file {read
        getattr execute open}
```

Since no corresponding process transition rule from the `untrusted_app` domain to the `tee` domain via `tee_exec` file is defined, and no `execute_no_trans` access type is granted, a process running in the `untrusted_app` domain cannot execute a file labeled as `tee_exec`. There are two plausible explanations. One is the use of tools like `audit2allow` (SELinux Project, 2014) to automatically generate rules, as discussed above. The other is the failure to clean up rules that were tested at some point but are no longer required. Below is an example of a vestigial rule that allows access to the debug interface of the Qualcomm KGSL GPU driver, which is itself disabled in production builds:

```
allow untrusted_app sysfs_kernel_debug_kgsl
        : file {read getattr}
```

3.2.4 Potentially Dangerous Rules

When working under tight time-to-market requirements, OEMs might decide to ship less strict security policies rather than make invasive changes to their codebase. This leads to a number of potentially dangerous rules appearing in OEM policies, like access to the `procfs` security-related filesystem objects. The rules below give `read/write` permissions on such objects to a trusted `hal` domain, a `release_app` domain and an `untrusted_app` domain.

```
allow hal proc_security : file {write
        getattr open}
```

```
allow release_app proc_security : file
    {ioctl read getattr lock open}
```

```
allow untrusted_app proc_security : file
        {read getattr open}
```

While processes running in the hal system domain can be considered trusted, the first rule is undesirable because it increases the attack surface of certain interfaces (like sensitive procfs settings) if the trusted process is compromised. The same applies to applications put in the release_app domain, as in the second rule. The third rule is even more dangerous, because it allows malicious applications running in the untrusted_app domain to get sensitive information, such as mmap_min_addr, memory randomization parameters and kernel pointer exposure settings, that can be used for further exploits.

Another example of a potentially dangerous rule is allowing processes running in the untrusted_app domain to read/write application data belonging to the system_app domain:

```
allow untrusted_app system_app_data_file :
        file {read write getattr}
```

However, since processes from untrusted_app and system_app domains will be run with different UIDS, the Linux DAC layer would guard against such arbitrary accesses unless a system application erroneously made its own files world-accessible.

In general, OEMs should have no additions to the set of rules for the untrusted_app domain, because any new allow rule increases the possible attack surface for malicious untrusted_app applications. However, Table 1 shows that almost all OEMs do add new rules for the untrusted_app domain.

On a positive note, OEMs are aware of such mistakes; some of them have been already fixed in the subsequent Android 5.1 update. The major reason behind these fixes was the release of the Android Compatibility Test Suite (CTS)[2] version 5.1, that added tests to ensure that AOSP neverallow rules are not violated by any process running on a device.

3.2.5 Discussion

We found several problematic patterns in the Android 5.0 OEM SEAndroid policies we examined. We conjecture that the reason for their presence is the relative unfamiliarity with SEAndroid. Google utilizes the set of neverallow rules in order to try to prevent OEMs from making security mistakes. However, while this approach might prevent some mistakes, it can also create difficulties for OEMs. For example any Global Platform-enabled TEE design [3] will likely end up with untrusted_app applications needing to access a kernel driver for their memory referencing. In Android 5.1 this conflicts with the existing neverallow rules, and as a result OEMs are forced to come up with a workaround. In the next section we propose a set of tools that can further help OEMs to avoid security mistakes and at the same time do not imply any restrictions on OEMs.

[2] source.android.com/compatibility/cts/index.html

[3] globalplatform.org

4 NEW TOOLS FOR SEANDROID

Although our systematic manual analysis unearthed some problem areas in the policies we analyzed, the process was cumbersome and time-consuming using the currently available tools. Based on our experience, we argue that new tools or new functionality in existing tools are necessary to aid both OEMs and security researchers to create and analyze SEAndroid policies effectively. We identify several such desirable tools below. We have implemented the first on the list (live policy analyzer) and are working on the rest.

4.1 Live Policy Analyzer

Existing SELinux policy analysis tools focus solely on the policy itself, and do not address the question of how the policy rules apply to a specific target device. A tool that can answer questions like "what files can a specified process on a device access?" or "what processes on a device can access a specified file?" would be very useful for the analyst. We developed SEAndroid Live device analysis tool (SEAL)[4] for this purpose. SEAL allows different queries that take into account not only the SEAndroid policy loaded on the device, but also the actual device state, i.e. running processes and filesystem objects. SEAL offers command line and GUI interfaces, and queries the device over the adb interface. In order to obtain results about the entire device filesystem, the target device has to be either rooted or running an engineering build. Figure 1 shows the architecture of SEAL.

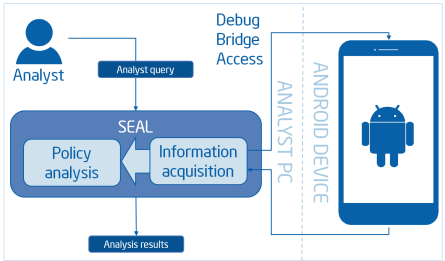


Figure 1: SEAL tool architecture

[4] github.com/seandroid-analytics/seal

4.2 Policy Decompilation Tool

One of the main problems during our manual analysis was the lack of a tool to easily identify and analyze changes that an OEM made to the default AOSP policy. Current tools like `apol` and `sediff` are not directly suited for this task, as described in Section 3.2. It would be very beneficial to have a tool able to transform a binary policy into a set of source files organized similarly to the AOSP SEAndroid source policy. In this case, it would not only be possible to perform manual analysis in a more organized and convenient manner, but also to employ standard text manipulation tools to compare or filter needed information.

4.3 Policy Visualization Tool

SELinux has the notion of `attributes`, that allow organizing policy types and domains in a hierarchical manner. This is a very powerful mechanism that can easily be misconfigured by mistake. The data collected in section 3.1 showed that most OEMs do not create new policy attributes, perhaps due to the perceived complexity of the attribute mechanism. A tool to visualize hierarchies induced by attributes may help an analyst better understand and make use of attributes effectively.

4.4 Policy Analyzer

The hardest part of our manual analysis was identifying rules that are either potentially dangerous or possibly unnecessary. The analysis can be automated with a set of heuristic checks. The tool can also utilize SEAL in order to make policy queries with regards to the device state.

Let us consider the following rule from section 3.2.4 and explain how it can be automatically detected as suspicious:

```
allow untrusted_app tee_exec : file {read
        getattr execute open}
```

The tool first uses SEAL to fetch from a device all files with the `tee_exec` label. Then it queries SEAL for all labels and DAC permissions of all higher-level directories on the path to each `tee_exec` file, and tries to determine if the `untrusted_app` domain can even reach the target file to perform the requested operations. If the first check passes, the analyzer can further check that each requested access control type makes sense. For example, in order for `execute` to succeed given that `execute_no_trans` access is not granted, there has to be a `type_transition` rule defined; furthermore, in order to be able to `write` or `read` a file, one would also normally need to have `open` permission. The policy analyzer can mark the rule as not functional if the checks fail.

In order to identify potentially dangerous rules, the policy analyzer can scan rules for possible additional usages of default types, mentioned in Section 3.2.1, and analyze new rules associated with sensitive types, such as `tee` or `proc_security`, or untrusted domains, such as the `untrusted_app` domain.

One use of the policy analyzer is its integration into OEMs' automatic build systems, in order to consistently verify that the policy does not contain unreachable or potentially dangerous rules, and that it is optimized with regards to the usage of attributes and types. This would provide value for OEMs, since policy additions might be made by different development teams, possibly without detailed knowledge of SEAndroid. The output of the tool can be further analyzed manually by a person with detailed knowledge of SEAndroid, in order to reject or accept suggested modifications.

5 RELATED WORK

Several tools and methods originally designed for SELinux are relevant to the new mobile environment.

The *de facto* standard for handling SELinux policies in text and binary format is the SETools library (Tresys, 2014): this contains the aforementioned `apol` and `sediff` tools, which can be used interchangeably on SELinux and SEAndroid.

Formal methods have been used for SELinux policy analysis. Gokyo (Jaeger et al., 2003) is a policy analysis tool designed to identify and resolve conflicting policy specifications. Usage of the HRU security model (Harrison et al., 1976) has been proposed as an approach to SELinux policy analysis (Amthor et al., 2011). Information flow analysis has been applied to SELinux policies (Guttman et al., 2005). These analysis methods are not SELinux-specific, and can be easily adapted to SEAndroid.

Some researchers have applied information visualization techniques to SELinux policy analysis (Clemente et al., 2012), also in combination with clustering (Marouf and Shehab, 2011). These techniques are also system-agnostic, and we may use them in future SEAndroid tools.

SELinux policy generation and refining tools are rare. Polgen, a tool for semi-automated SELinux policy generation based on system call tracing (Sniffen et al., 2006), appears to be no longer in active development. The SELinux userspace tools (SELinux Project, 2014) can generate SELinux policies. One

of these tools, `audit2allow`, is widely used to automatically generate and refine SELinux policies by converting SELinux audit messages into rules; these policies, however, are not necessarily correct, complete or secure, since the rules depend on code paths taken during execution, and there is no way to distinguish intended and possibly malicious application behavior. These tools are used both in SELinux and SEAndroid.

There has been some research in applying Domain Specific Languages (DSL) (Fowler, 2010) to SELinux policy development and verification (Hurd et al., 2009). The authors proposed a tool (`shrimp`) to analyze and find errors in the SELinux Reference Policy, similar to the `Lint` tool for C. This is similar to a tool we propose, but different in scope as it is limited to analysis of the SELinux reference policy.

The only SEAndroid-specific analysis method is based on audit log analysis with machine learning (Wang et al., 2015). This approach is completely different from what we propose, since it relies on significant volumes of data to classify rules.

6 CONCLUSIONS

In this paper we presented a number of common mistakes made by OEMs in their SEAndroid policies, suggesting potential reasons behind them. As a result of this study, we identified a number of practical tools that should help OEMs and security researchers to improve SEAndroid policies. We provided the implementation of a first tool, SEAL, and we are currently working on the rest.

REFERENCES

Amthor, P., Kuhnhauser, W., and Polck, A. (2011). Model-based safety analysis of selinux security policies. In *NSS*, pages 208–215. IEEE.

Badger, L., Sterne, D., Sherman, D., Walker, K., Haghighat, S., et al. (1995). Practical domain and type enforcement for UNIX. In *Security and Privacy*, pages 66–77. IEEE.

Bauer, M. (2006). Paranoid penguin: an introduction to Novell AppArmor. *Linux Journal*, (148):13.

Clemente, P., Kaba, B., Rouzaud-Cornabas, J., Alexandre, M., and Aujay, G. (2012). Sptrack: Visual analysis of information flows within selinux policies and attack logs. In *AMT*, pages 596–605. Springer.

Fowler, M. (2010). *Domain-specific languages*. Pearson Education.

Guttman, J. D., Herzog, A. L., Ramsdell, J. D., and Skorupka, C. W. (2005). Verifying information flow goals in security-enhanced Linux. *Journal of Computer Security*, 13(1):115–134.

Harrison, M. A., Ruzzo, W. L., and Ullman, J. D. (1976). Protection in operating systems. *Commun. ACM*, 19(8):461–471.

Hurd, J., Carlsson, M., Finne, S., Letner, B., Stanley, J., and White, P. (2009). Policy DSL: High-level Specifications of Information Flows for Security Policies.

Jaeger, T., Sailer, R., and Zhang, X. (2003). Analyzing integrity protection in the selinux example policy. In *USENIX Security*, page 5.

Marouf, S. and Shehab, M. (2011). SEGrapher: Visualization-based SELinux policy analysis. In *SAFECONFIG*, pages 1–8. IEEE.

Schaufler, C. (2008). Smack in embedded computing. In *Ottawa Linux Symposium*.

SELinux Project (2014). Userspace tools. `https://github.com/SELinuxProject/selinux/wiki`. Accessed: 2015-09-29.

Smalley, S. and Craig, R. (2013). Security Enhanced (SE) Android: Bringing flexible MAC to Android. In *NDSS*, volume 310, pages 20–38.

Smalley, S., Vance, C., and Salamon, W. (2001). Implementing SELinux as a Linux security module. *NAI Labs Report*, 1(43):139.

Sniffen, B. T., Harris, D. R., and Ramsdell, J. D. (2006). Guided policy generation for application authors. In *SELinux Symposium*.

Spencer, R., Smalley, S., Loscocco, P., Hibler, M., and Lepreau, J. (1999). The Flask security architecture: System support for diverse policies. In *USENIX Security*.

Tresys (2014). SETools project page. `https://github.com/TresysTechnology/setools3/wiki`. Accessed: 2015-09-29.

Wang, R., Enck, W., Reeves, D., Zhang, X., Ning, P., Xu, D., Zhou, W., and Azab, A. (2015). EASE-Android: Automatic Policy Analysis and Refinement for Security Enhanced Android via Large-Scale Semi-Supervised Learning. In *USENIX Security*.

Zhou, Y. and Jiang, X. (2012). Dissecting android malware: Characterization and evolution. In *Security and Privacy*, pages 95–109. IEEE.

Publication III

Reshetova, Elena and Bonazzi, Filippo and and Asokan, N. SELint: an SE-Android policy analysis tool. In *ICISSP*, Location, pages, and other detailed information, October 2017.

SELint: an SEAndroid policy analysis tool

Elena Reshetova[1], Filippo Bonazzi[2], N. Asokan[2,3]
[1] *Intel OTC, Helsinki, Finland*
[2] *Aalto University, Helsinki, Finland*
[3] *University of Helsinki, Helsinki, Finland*
elena.reshetova@intel.com, filippo.bonazzi@aalto.fi, asokan@acm.org

Keywords: Security, SEAndroid, SELinux, Android, Access Control, policy analysis

Abstract: SEAndroid enforcement is now mandatory for Android devices. In order to provide the desired level of security for their products, Android OEMs need to be able to minimize their mistakes in writing SEAndroid policies. However, existing SEAndroid and SELinux tools are not very useful for this purpose. It has been shown that SEAndroid policies found in commercially available devices by multiple manufacturers contain mistakes and redundancies. In this paper we present a new tool, SELint, which aims to help OEMs to produce better SEAndroid policies. SELint is extensible and configurable to suit the needs of different OEMs. It is provided with a default configuration based on the AOSP SEAndroid policy, but can be customized by OEMs.

1 INTRODUCTION

During the past decade Android OS has become one of the most common mobile operating systems. However, at the same time we have seen a big increase in the number of malware and exploits available for it (Zhou and Jiang, 2012; Smalley and Craig, 2013). Many classical Android exploits, such as GingerBreak and Exploid, attempted to target system daemons that ran with elevated - often unlimited - privileges. A successful compromise of such daemons results in the compromise of the whole Android OS, and the attacker would be able to obtain permanent root privileges on the device. Initially Android relied only on its permission system, based on Linux Discretionary Access Control (DAC), to provide security boundaries. However, after it became evident that DAC cannot protect from such exploits, a new Mandatory Access Control (MAC) mechanism has been introduced. SEAndroid (Smalley and Craig, 2013) is an Android port of the well-established SELinux MAC mechanism (Smalley et al., 2001) with some Android-specific additions. Similarly to SELinux, SEAndroid enforces a system-wide policy. The default SEAndroid policy was created from scratch and is maintained as part of the Android Open Source Project (AOSP)[1]. Starting from the 5.0 Lollipop release, Android requires every process to run inside a confined SEAndroid domain with a proper set of access control rules defined. This has forced many Android Original Equipment Manufacturers (OEMs) to hastily define the set of access control domains and associated rules needed for their devices. Our recent study (Reshetova et al., 2016) showed that all OEMs modify the default SEAndroid policy provided by AOSP due to many customizations implemented in their Android devices. The difficulty of writing well-designed SELinux policies together with high time-to-market pressure can possibly lead to the introduction of mistakes and major vulnerabilities. The study classified common mistake patterns present in most OEM policies and concluded that new practical tools are needed in order to help OEMs avoid these mistakes. In this paper we make the following contributions:

- Design of **a new, extensible tool, SELint,** that aims to help Android OEMs to overcome common challenges when writing SEAndroid policies (Section 4). In contrast to existing SELinux and SEAndroid tools (described in Section 3), it can be used by a person without a deep understanding of SEAndroid, given the initial configuration by an expert. The community can write new analysis modules for SELint in the form of SELint plugins. This is especially important given that the SEAndroid policy format changes with every release, and new notions and mechanisms are introduced by Google.

[1] source.android.com

- **An initial configuration** for SELint, based on the AOSP SEAndroid policy, which **was found to be useful** by the SEAndroid community in our evaluation of SELint (Section 5.1).
- A full **implementation** of SELint that that **fits OEM policy development workflows**, providing **reasonable performance** and allowing **easy customization** by OEMs (Section 5).

2 Background

2.1 SELinux and SEAndroid

SELinux (Smalley et al., 2001) was the first mainline MAC mechanism available for Linux-based distributions. Compared to other mainline MAC mechanisms present today in the Linux kernel, it is considered to be the most fine-grained and the most difficult to understand and manage due to the lack of a minimal policy (like in Smack (Schaufler, 2008)) or a learning mode (like in AppArmor (Bauer, 2006)). Despite this, it is enabled by default in Red Hat Enterprise Linux (RHEL) and Fedora with pre-defined security policies.

The core part of SELinux is its Domain/Type Enforcement (Badger et al., 1995) mechanism, which assigns a `domain` to each subject, and a `type` to each object in the system. A subject running in `domain` can only access an object belonging to `type` if there is an `allow` rule in the policy of the following form:

```
allow domain type :  class permissions
```

where `class` represents the nature of an object such as file, socket or property, and `permissions` represent the kinds of operations being permitted on this object, like `read`, `write`, `bind` etc.

The **SEAndroid** (Smalley and Craig, 2013) MAC mechanism is mostly based on SELinux code with some additions to support Android-specific mechanisms, such as the Binder Inter Process Communication (IPC) framework. However, SEAndroid's policy is fully written from scratch and is very different from SELinux's reference policy. AOSP predefines a set of application domains, like `system_app`, `platform_app` and `untrusted_app`; applications are assigned to these domains based on the signature of the Android application package file (`.apk`). Other services and processes are assigned to their respective domains based on filesystem labeling or direct domain declaration in the service definition in the `init.rc` file. One notable feature of the SEAndroid policy is active usage of predefined M4 macros that make the policy more readable and compact. For example, the `global_macros` file defines a number of M4 macros that denote sets of typical permissions needed for common classes, such as `r_file_perms` or `w_dir_perms`. Another example is the `te_macros` file, that provides a number of M4 macros used to combine sets of rules commonly used together.

2.2 SEAndroid OEM Challenges

The SEAndroid reference policy only covers default AOSP services and applications. Therefore, highly customized OEM Android devices require extensive policy additions.

Our already mentioned study of different OEM SEAndroid policies for Android 5.0 Lollipop (Reshetova et al., 2016) showed that most OEMs made a significant number of additions to the default AOSP reference policy. The biggest changes are the additions of new types and domains, as well as new `allow` rules. The study also identified a number of common patterns that most OEM policies seem to follow:

- **Overuse of default types**. SEAndroid declares a set of default types that are assigned to different objects unless a dedicated type is defined in the policy. Most OEMs leave many such types in their policies, which indicates a use of automatic policy creation tools such as `audit2allow` (SELinux, 2014).
- **Overuse of predefined domains**. OEMs do not typically define dedicated domains for their system applications, but tend to assign these applications to predefined `platform_app` or `system_app` domains. This creates over-permissive application domains and violates the principle of least privilege.
- **Forgotten or seemingly useless rules**. OEM policies have many rules that seem to have no effect. This might be due to an automatic rule generation or a failure to clean up unnecessary rules that were no longer required.
- **Potentially dangerous rules**. A number of potentially dangerous rules can be seem in some OEM policies, including granting additional permissions to `untrusted_app` domain. This might be due to lack of time to adjust their service or application implementation to minimize security risks or due to inability to identify some rules as being dangerous.

3 RELATED WORK

Since SELinux existed on its own long before SEAndroid, most of the available tools are designed to handle and analyze SELinux policies. They can be used for SEAndroid but they don't take specific aspects of SEAndroid policies into account. This makes it challenging for OEMs to use existing tools to detect the problems outlined in Section 2.2. For example, in order to determine if the policy contains potentially dangerous rules, it is very important to understand the semantics of SEAndroid types and policy structure - an ability which all existing SELinux tools lack. Moreover, even the small group of SEAndroid tools described in Section 3.2 does not address the challenges described in Section 2.2.

3.1 SELinux Tools

SETools (Tresys, 2016) is the official collection of tools for handling SELinux policies in text and binary format. Some of its tools, like `apol`, are suitable for formal policy analysis, for example for flow-control analysis. Others allow policy queries and policy parsing and as such it can be used on both SELinux and SEAndroid. An important part of SETools is a policy representation library which is used in both SEAL and SELint.

Formal methods have been applied to SELinux policy analysis. Gokyo (Jaeger et al., 2003) is a tool designed to find and resolve conflicting policy specifications. Guttman *et al.* (Guttman et al., 2005) applies information flow analysis to SELinux policies. The HRU security model (Harrison et al., 1976) has been used to analyze SELinux policies (Amthor et al., 2011). Hurd *et al.* (Hurd et al., 2009) applied Domain Specific Languages (DSL) (Fowler, 2010) in order to develop and verify the SELinux policy. The resulting tool, `shrimp`, can be used to analyze and find errors in the SELinux Reference Policy. Information visualization techniques have been applied to SELinux policy analysis in (Clemente et al., 2012), also in combination with clustering of policy elements (Marouf and Shehab, 2011). These analysis methods are largely academic, and no practical tools based on them are used by the SELinux community.

Polgen (Sniffen et al., 2006) is a tool for semi-automated SELinux policy generation based on system call tracing. Unfortunately it appears to be no longer in active development. SELinux also provides a set of userspace tools (SELinux, 2014) that can be used on both SELinux and SEAndroid. One of these tools, `audit2allow`, is widely used by Android OEMs to automatically generate and expand SEAndroid policies. The tool works by converting denial audit messages into rules based on a given binary policy. These rules, however, are not necessarily correct, complete or secure, since they entirely depend on code paths taken during execution and require a good understanding of the software components involved, as well as on the correct labeling of subjects and objects in the system. Furthermore, automatically-generated rules fail to use high-level SEAndroid policy features such as attributes and M4 macros: this results in comparatively less readable policies.

3.2 SEAndroid Tools

Our aforementioned study (Reshetova et al., 2016) presented SEAL, an SEAndroid live device analysis tool. SEAL works with a real or emulated Android device over the Android Debug Bridge (ADB); it can perform different queries that take into account not only the binary SEAndroid policy loaded on the device, but also the actual device state, i.e. running processes and filesystem objects. The EASEAndroid policy refinement method is based on audit log analysis with machine learning (Wang et al., 2015). This approach is completely different from what we propose, since it relies on significant volumes of data to classify rules. Unfortunately, it is very hard to obtain this volume of data, since it would require collecting log files from millions of Android devices with possible privacy implications. The most recent SEAndroid policy analysis and refinement tool is SEAndroid Policy Knowledge Engine (SPOKE) (Wang, 2016). It automatically extracts domain knowledge about the Android system from application functional tests, and applies this knowledge to analyze and highlight potentially over-permissive policy rules. SPOKE can be used to identify new heuristics that can be implemented as new SELint plugins. The downside of SPOKE is reliance on application functional tests, which are often incomplete, and the fact that it cannot be easily integrated into the standard development workflow.

4 SELint

4.1 Requirements

We identify the following generic requirements that a tool like SELint must fulfill.

R 1. Source policy-based. The existing tool landscape presented in Section 3 does not feature any tool able to perform semantic analysis on source SEAndroid policies. Since Android OEMs work on source

SEAndroid policies as part of their Android trees, the tool needs to work with source SEAndroid policies.

R 2. Configurable by experts, usable by all. Existing tools require extensive domain knowledge to be used. Since building such a knowledge takes considerable time, it might be challenging for OEMs to have all of their development team trained appropriately. We intended for our tool to fit into an Android OEM policy development workflow, where many developers, overseen by one or a few experienced SEAndroid analysts, contribute small changes to the policy. Therefore, it must be possible for an experienced analyst to configure the tool ahead of time, and provide a ready-to-run tool to regular developers, who can simply run the tool on their policy modifications and verify that no issues are highlighted.

R 3. Reasonable performance. Since we are targeting inclusion into an Android OEM workflow, the tool must have reasonable time and memory performance; this is necessary for the tool to be used as part of the build toolchain, or even more appropriately when committing changes using the OEM's version control software (VCS).

R 4. Easy to configure and extend. Finally, targeting the wide community of Android OEMs makes it impossible to know in advance all possible use cases and requirements, present and future. It is our objective to allow analysts to implement their own analysis functionality and embed their domain knowledge into the tool. For this reason, the tool must be easily configurable and extensible by the community.

4.2 General Architecture and Implementation

To meet Requirement 4 stated in section 4.1, we designed SELint following a plugin architecture. The goal of such an architecture is to support custom third-party analysis plugins that any community member can create. The core part of SELint is responsible for processing the source SEAndroid policy. The *SELint core* takes care of handling user input, such as command line options and configuration files. After the source policy has been parsed, its representation is given to the *SELint plugins* which perform the actual analysis. We have developed an initial set of plugins, which provide generally useful functionality; interested Android OEMs can develop more plugins to implement their own analysis requirements.

The overall architecture is shown in Figure 1; the existing plugins are individually described in the following sections. The implementation of SELint and the existing plugins are released under the Apache License 2.0, which allows the community to freely use and modify the software. The `policysource` library is released under the GNU Lesser General Public License v2.1.

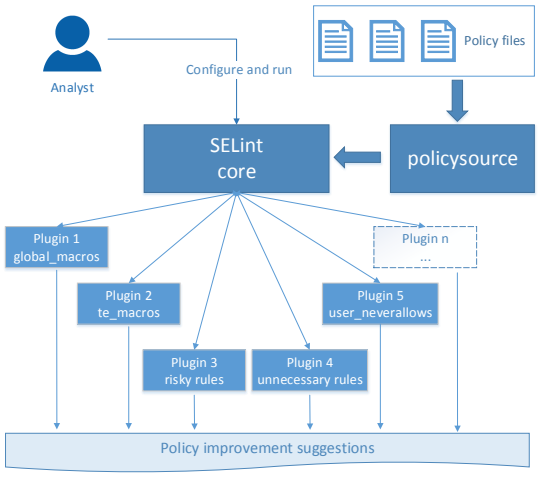


Figure 1: The architecture of SELint

The SELint executable and all plugins have an associated configuration file. This allows policy experts to adapt each plugin to the semantics of their own policies, for example to define OEM-specific policy types. This way, SELint can be run with different preset "*profiles*" specifying different options, policy configurations and requested analysis functionalities. The following sections describe each existing SELint plugin in detail.

4.3 Plugin 1: Simple Macros

Goal As mentioned in Section 2.1, using M4 macros

```
r_file_perms → { getattr open read ioctl lock }
```

Figure 2: A `global_macros` definition and expansion

where applicable is a non-functional requirement of SEAndroid policy development: while not affecting policy behavior, their use makes for a more compact and readable policy. The first type of M4 macros extensively used in SEAndroid policies is a simple text replacement macro, without arguments, that is used to represent sets of related permissions. Such macros are defined in the `global_macros` file in the SEAndroid policy source files. An example of such macro is shown in Figure 2. The Simple macro plugin scans the policy for rules granting sets of individual permissions which could be represented in a more compact way by using an existing `global_macros` macro; it then suggests replacing the individual permissions with an usage of said macro.

Implementation The plugin looks for rules which specify individual permissions whose combination is equivalent to the expansion of a `global_macros` macro. It then suggests rewriting said rules, replacing the individual permissions with the unexpanded macro. An example is shown in Figure 3. For this particular case, the plugin suggest replacing a set of permissions {`gettattr open read search ioctl`} with a macro `r_dir_perms`. Permissions not contained in the macro (in this case `create`) are still specified individually in the final rule. The plugin can suggest both full matches (for rules that grant 100% of the permissions contained in a macro) and partial matches above a threshold (for rules that grant at least X% of the permissions contained in a macro). This threshold is a user-defined parameter, specified in the plugin configuration file; we assigned it a default value of 0.8 (80%).

Rule:
```
allow logd rootfs:dir
 { getattr create open read search ioctl };
```

Macro:
```
r_dir_perms → { open getattr read search ioctl }
```

Suggestion:
```
allow logd rootfs:dir { r_dir_perms create };
```

Figure 3: An example usage of the Simple macro plugin

Limitations The plugin only deals with simple, static macros without arguments. Dynamic macros such as those defined in the `te_macros` file are handled by the dedicated plugin described in the next section.

4.4 Plugin 2: Parametrized Macros

Goal Another commonly used set of M4 macros includes more complex, dynamic M4 macros with multiple arguments. Such macros are mainly used to group rules which are commonly used together; their expansion can in turn contain other macros. In SE-Android policies, such macros are defined in the `te_macros` file. An example is shown in Figure 4. Similarly to the previous, this plugin detects existing macro definitions, and suggests new usages.

```
        'file_type_trans($1, $2, $3)'
                        ↓
'allow $1 $2:dir ra_dir_perms;
 allow $1 $3:dir create_dir_perms;
 allow $1 $3:notdevfile_class_set
                              create_file_perms;'
```

Figure 4: A `te_macros` macro definition and expansion with arguments.

Implementation The plugin looks for sets of individually specified rules whose combination is equivalent to the expansion of a `te_macros` macro with some set of arguments. It then suggests substituting said rules with a usage of the unexpanded macro with the proper arguments. An example is shown in Figure 5: the plugin finds the existing macro which expands into the given set of rules - in this case, `unix_socket_connect`. It then extracts the arguments from the rules: $1 is "a", $2 is "b" and $3 is "c". The result is a suggestion for substituting the two rules with the macro usage. The plugin can suggest both full matches (for sets of rules that match 100% of the rules contained in a macro expansion) and partial matches above a user-defined threshold. Its default value is 0.8 (80%).

Rules:
```
allow a b_socket:sock_file write;
allow a c:unix_stream_socket connectto;
```

Macro:
```
unix_socket_connect($1, $2, $3)
                ↓
allow $1 $2_socket:sock_file write;
allow $1 $3:unix_stream_socket connectto;
```

Suggestion:
```
unix_socket_connect(a, b, c)
```

Figure 5: An example usage of the Parametrized macro plugin.

Limitations The problem of detecting sets of rules that match possible macro expansions can be transformed into a variant of the knapsack problem (Kellerer et al., 2004), namely a multidimensional knapsack problem. In our case, the knapsack capacity is the number of arguments a macro can have, and the knapsack items are the possible values of these arguments; the knapsack is multidimensional because filling an argument does not affect the available capacity for the others. Instead of finding the single most profitable combination of argument-values, our objective is to find all the combinations of argument-values which, used as arguments in as many macro expansions, produce sets of rules entirely or partially (above a threshold) contained in the policy. The problem can be formalized as:
For each macro m, find all combinations of values for arguments \$1, \$2 and \$3 such that y is above an user-given threshold t. y is computed as: $y = score(m(i,j,k))$, subject to $i \in N_i, j \in N_j|_i, k \in N_k|_{ij}$, where N_i is the set of possible values of \$1, $N_j|_i$ is the set of possible values of \$2 given i as \$1, and $N_k|_{ij}$ is the set of possible values of \$3 given i as \$1 and

j as \$2. $score(m(i,j,k))$ is the score of the macro expanded with the arguments i, j and k: the score of a macro expansion is given by the number of its rules actually found in the policy divided by its overall number of rules.

The multidimensional knapsack optimization problem is known to be NP-hard (Magazine and Chern, 1984), and it has various approximate solutions (Chu and Beasley, 1998; Hanafi and Freville, 1998). In our case, the problem quantities are the number of arguments a macro can have (existing macros have 1-3), the number of rules a macro expansion can produce (existing macros have 1-7), and the number of values a macro argument can have (in principle infinite, in practice dependent on the policy, usually in the thousands). In practice, the number of rules (#2) tends to increase linearly with the number of arguments (#1). This is due to the fact that macros with more arguments can define more complex behavior, which tends to be described in more rules.

As a first implementation, we realized a simple solution based on exploration of the solution space: we try to aggregate all the policy rules into sets corresponding to macro expansions. The problem quantities described above result in a significant time expenditure required to explore the whole solution space: therefore, as we discuss in Section 5.2, this plugin takes considerably more time than all others.

4.5 Plugin 3: Risky Rules

Experts analyzing OEM modifications to SEAndroid policies often use certain heuristics. The analysis usually starts from the list of AOSP SEAndroid domains and types that are more likely to cause potential vulnerabilities in OEM policies. The most common are:

- **Untrusted domains.** Some domains are intended to run potentially malicious code, such as `untrusted_app`, and therefore their privileges are designed to be minimal. Any additional `allow` rules created by OEMs for such domains are suspicious and need to be analyzed.
- **Trusted Computing Base (TCB) domains and types.** The AOSP policy has several core domains and types, which form its TCB. The processes that run in these domains are provided by AOSP, and so are the minimal required policy rules. Sometimes, OEMs have to create additional rules for some of these domains: however, since doing so increases the chance of compromising the TCB, such rules need thoughtful inspection.
- **Security-related domains and types.** Special attention must be paid to AOSP domains and types directly related to system security, such as the `tee` domain or the `proc_security` type. Mistakes in additional `allow` rules for these domains and types can lead to a direct loss of system security.

An analyst usually checks an OEM policy for additional rules where the above domains or types are present, and then manually inspects each rule analysing its domain, type and permissions to determine if the rule is actually risky. This process is tedious, and most of the time is spent just finding the rules which need special attention. To help analysts find these rules quickly, we developed the `risky_rules` plugin, which processes each rule and assigns it a score based on one of two criteria.

The first scoring criterion is based on *risk*. We define the **risk level** for rule components as the level of potential damage to the system caused by misuse of the component: security-sensitive components will have high risk scores, while generic components will have lower risk scores. Untrusted domains will have a high *risk* score as well, because we want to select any additional rules over such domains for manual inspection. Component risk level in turn determines the risk level of a rule, which is obtained by combining the risk levels of its components. The risk level of a rule is then defined as the level of potential damage to the system allowed by the rule. The risk score helps analysts to quickly obtain a prioritized list of policy rules which need manual inspection; this is especially useful when analysts have strict time constraints, and only have time to examine a limited number of rules. The *risk* scoring system is described in Section 4.5.1.

The second scoring criterion is based on *trust*. We define the **trust level** for rule components as a measure of closeness to the core of the system: key system components will have a high *trust* level, while user applications will have a low *trust* level. This in turn allows us to detect rules which cross trust boundaries, *e.g.* comprising a *high* component and a *low* component or vice versa. This scoring system is useful for an analyst as well, because it can quickly identify additional OEM rules which breach trust boundaries and select them for manual inspection. The *trust* scoring system is described in Section 4.5.2.

The desired scoring system can be specified in the plugin configuration file. We have provided an initial `risky_rules` plugin configuration based on our knowledge and experience with the AOSP policy. While our classification might be considered subjective, feedback discussed in Section 5.1 indicates that SEAndroid policy writers agree with our approach.

4.5.1 Measuring Risk

Goal As mentioned above, rules in a policy can have different risk levels, depending on the types they deal with and the permissions they grant. The *risk* scoring system of the `risky_rules` plugin assigns a score to every rule in the policy, prioritizing potentially riskier rules by assigning them higher scores.

Implementation The *risk* scoring system computes the overall score for a rule by evaluating its domain, type, and permissions or capabilities. The plugin configuration file defines partial *risk* scores for various rule elements. Relevant AOSP domains and types are grouped by risk level into "*bins*", which are assigned a partial *risk* score with a maximum of 30. When computing the score for a rule, the partial scores of its domain and type are added. We treat domains and types equally, because both the running process and data of a program might be equally important in evaluating how risky a rule is. For example, a process running in a security sensitive domain (*e.g.* `keystore`) should not accept any command from other processes running in unauthorized domains, because they might induce malicious changes in its execution flow. Similarly, other unauthorized processes should not be able to modify the configuration data of a security sensitive process (*e.g.* data labeled as `keystore_data`), for similar reasons. The initial set of bins and their default scores are depicted in Table 1.

Bin name	Example types	Risk
`user_app`	`untrusted_app`	30
`security_sensitive`	`tee, keystore, security_file`	30
`core_domains`	`vold, netd, rild`	15
`default_types`	`device, unlabeled, system_file`	30
`sensitive`	`graphic_device`	20

Table 1: `risky_rules` plugin default bins and partial *risk* scores.

The `user_app`, `core_domains` and `security_sensitive` bins match groups defined earlier in this section. `user_app` and `security_sensitive` have the maximum score of 30, while the score for `core_domains` is 15 due to less overall risk to the system. The `default_types` bin has a maximum score of 30, because it contains types that should not normally be used by OEMs and therefore likely indicate a mistake in a rule.

When computing the overall *risk* score for a rule, in addition to evaluating a rule's domain and type elements, the *risk* scoring system must also take its permissions and capabilities into account. In SEAndroid, permissions are meaningless in isolation, and only meaningful to determine risk when combined with the domain to which they are granted and the type over which they are granted: for this reason, we combine these when computing the *risk* score for a rule. We do this by assigning permissions a multiplicative coefficient instead of an additive partial score; the sum of domain and type score for a rule is multiplied by this coefficient. Commonly used permissions are categorized by level of risk into three groups, `perms_high`, `perms_med` and `perms_low`: each group is assigned a coefficient based on the sensitivity of its permissions, with a maximum of 1. The sum of domain and type score is multiplied by the coefficient of the highest set which contains permissions granted by the rule; this is done because we are interested in determining the upper bound of risk for a rule. Table 2 shows the groups, permissions and default values of coefficients.

Set name	Example permissions	Coefficient
`perms_high`	`ioctl, write, execute`	1
`perms_med`	`read, use, fork`	0.9
`perms_low`	`search, getattr, lock`	0.5

Table 2: `risky_rules` plugin default permission sets and coefficients.

Capabilities are treated differently from permissions. In SEAndroid, capabilities are granted by a domain to itself, and - unlike permissions - are meaningful on their own: they have the same effect on the system regardless of the domain they are granted to. For example, the following rule grants the `vold` daemon the `CAP_CHROOT` capability, which allows it to perform the `chroot` system call:

```
allow vold self:capability sys_chroot;
```

We do not divide capabilities into separate groups: this is due to the fact that, in Linux, capabilities are commonly believed to be very hard to categorize as more or less dangerous, because of the consequences they can have on the system[2]. Since in SEAndroid capabilities are granted by a domain to itself, the target type in such a rule does not convey any additional information: therefore, we use a special scoring formula for rules granting capabilities. Capabilities are handled as types, and any capability is assigned the maximum score for a type (30): this score is added to the domain score to obtain the rule score.

The *risk* scoring system scores rules by their potential level of risk between 0 and 1, with maximum risk given a score of 1. As discussed above, risk scores are assigned to rules depending on the type of rule: the precise formulas are presented in Figure 6.

An example is shown in Figure 7. The first rule contains `untrusted_app` and `security_file`, which are both high-risk types (`user_app` and

[2] forums.grsecurity.net/viewtopic.php?f=7&t=2522

Allow rules granting permissions :

$$\text{score}_{risk}(rule) = \frac{\text{score}_{risk}(domain)+\text{score}_{risk}(type)}{M} \cdot C$$

$$C = \max_{0 \le i < nperms}(\text{coefficient}_{risk}(perm_i))$$

Allow rules granting capabilities:

$$\text{score}_{risk}(rule) = \frac{\text{score}_{risk}(domain)+\text{score}_{risk}(capabilities)}{M}$$

Type_transition rules:

$$\text{score}_{risk}(rule) = \frac{\text{score}_{risk}(domain)+\text{score}_{risk}(type)}{M}$$

M is the maximum value of the numerator (60), used to normalize the score between 0 and 1.

Figure 6: The *risk* scoring formulas for the `risky_rules` plugin.

`security_sensitive` respectively); however, the rule only grants the `getattr` and `search` permissions, which are two low-risk permissions. Thus, the rule has a medium *risk* score that in this case equals to 0.5. The second rule contains `untrusted_app` and `system_file`, which are both high-risk types (`user_app` and `default_types` respectively); furthermore, the rule grants the `execute` permission, which is a high-risk permission. Thus, the rule has a high *risk* score that in this case equals to 1.

```
0.50: .../domain.te:154: allow untrusted_app
        security_file:dir { getattr search };
1.00: .../domain.te:104:
    allow untrusted_app system_file:file execute;
```

Figure 7: An example of the `risky_rules` plugin with the *risk* scoring system.

4.5.2 Measuring Trust

Goal Rules in a policy can contain domains and types with different *trust* levels. Analysts usually inspect a policy by manually looking for rules which cross *trust* boundaries and making sure they are justified: this process is time-consuming and can be error prone. The *trust* scoring system of the `risky_rules` plugin automates this search: it assigns a score to every rule in the policy, prioritizing rules which cross *trust* boundaries by assigning them higher scores.

Implementation The *trust* scoring system combines the partial scores of domain and type in a rule to assign it an overall score. The plugin configuration file defines partial *trust* scores for various rule elements. AOSP domains and types are grouped into "*bins*", which are assigned a *trust* score with a maximum of 30. When computing the score for a rule, the partial scores of its domain and type are added. The initial bins with their default scores are depicted in Table 3.

Bin name	Example types	Trust
`user_app`	`untrusted_app`	0
`security_sensitive`	`tee, keystore, security_file`	30
`core_domains`	`vold, netd, rild`	20
`default_types`	`device, unlabeled, system_file`	5
`sensitive`	`graphic_device`	10

Table 3: `risky_rules` plugin default bins and partial *trust* scores.

For example, the `user_app` bin contains types assigned to generic user applications, such as `untrusted_app`; since user applications are not trusted, the *trust* score for this bin is minimum (0). The `security_sensitive` bin contains types assigned to data or components that have direct security impact, such as `tee`, `keystore`, `proc_security` *etc*. These components and their data are also highly trusted, since they form the TCB of the system, and therefore their *trust* score is maximum (30). The *trust* scoring system scores rules by the level of trust of their domain and type, regardless of the type of rule. Permissions and capabilities are ignored when computing the *trust* score for a rule. The level of trust can be *high* or *low*, giving place to 4 different scoring criteria: *trust_hl*, where the rule features a *high* domain and a *low* type, *trust_lh*, where the domain is *low* and the type is *high*, *trust_hh*, where both are *high*, and *trust_ll*, where both are *low*. The various *trust* criteria score rules between 0 and 1, where a score of 1 indicates that a rule is closest to the specified criterion. A high rule score is obtained naturally when looking for *high* components: to obtain a high rule score when looking for *low* components, the component partial score is subtracted from the maximum partial score before normalizing. Trust scores are assigned to rules using the formulas presented in Figure 8.

An example of one of the *trust* scoring systems (*trust_lh*) is shown in Figure 9. The first rule contains `untrusted_app`, which is a low-trust domain, and `system_file`, which is a low-trust domain. The scoring criterion assigns the maximum score to rules with a *low* domain and a *high* type: therefore, the rule has a medium *trust_lh* score, which in this case is 0.58. The second rule contains `untrusted_app`, which is a low-trust domain, and `security_file`, which is a high-trust type. According to the selected scoring criterion, the rule has the maximum *trust_lh* score of 1.

Trust_ll:

$$\text{score}_{trust}(rule) = \frac{(\frac{M}{2} - \text{score}_{trust}(domain)) + (\frac{M}{2} - \text{score}_{trust}(type))}{M}$$

Trust_lh:

$$\text{score}_{trust}(rule) = \frac{(\frac{M}{2} - \text{score}_{trust}(domain)) + (\text{score}_{trust}(type))}{M}$$

Trust_hl:

$$\text{score}_{trust}(rule) = \frac{(\text{score}_{trust}(domain)) + (\frac{M}{2} - \text{score}_{trust}(type))}{M}$$

Trust_hh:

$$\text{score}_{trust}(rule) = \frac{\text{score}_{trust}(domain) + \text{score}_{trust}(type)}{M}$$

M is the maximum value of the numerator (60), used to normalize the score between 0 and 1.

Figure 8: The *trust* scoring formulas for the `risky_rules` plugin.

```
0.58: .../domain.te:104:
    allow untrusted_app system_file:file execute;
1.00: .../domain.te:154: allow untrusted_app
        security_file:dir { getattr search };
```

Figure 9: An example of the `risky_rules` plugin with the *trust_lh* scoring system.

4.5.3 Limitations

Both scoring systems, *risk* and *trust*, assign a score to a rule by computing a formula over the partial scores of various rule elements. These partial scores must be defined by an analyst in the plugin configuration file, and simply reflect what an analyst is most interested in. Only the analyst who defined an element in the policy has the relevant knowledge to assign it a *risk* or *trust* score. A high rule score does not mean that a rule is dangerous, and a low score does not mean that a rule is safe: a high score represents a rule which the analyst deems more interesting, and vice versa.

4.6 Plugin 4: Unnecessary Rules

Goal Some rules are effective only when used in combination. For example, a `type_transition` rule is useless without the related `allow` rules actually enabling the requested access. Similarly, some permissions are meaningful only when granted in combination. For example, an `allow` rule which grants `read` on a file type, without granting `open` on the same type or `use` on the related file descriptor type, will not actually allow the file to be read. Another example is debug rules, which are effective only when used for an OEM internal engineering build, and should not be present in the derived user build which is actually shipped. An analyst may want to check that all such rules are correctly wrapped inside debug M4 macros, which prevent them from appearing in the final user build. The `unnecessary_rules` plugin searches the policy for rules which are ineffective or unnecessary, as in the examples above. It also looks for debug rules mistakenly visible in the user policy.

Tuple:

```
type_transition $ARG0 $ARG1:file $ARG2;
allow $ARG0 $ARG1:dir { search write };
allow $ARG0 $ARG2:file { create write };
```

If found:

```
type_transition a b:file c;
```

Look for:

```
allow a b:dir { search write };
allow a c:file { create write };
```

Figure 10: An example of the "ineffective rule combinations" functionality of the `unnecessary_rules` plugin.

Implementation The plugin provides 3 features: detection of ineffective rule combinations, detection of debug rules, and detection of ineffective permissions.

Ineffective rule combinations: The plugin detects missing rules from an ordered tuple of rules. Tuples can be specified by an analyst in the plugin configuration file, and can contain placeholder arguments. This functionality looks for rules matching the first rule in a tuple, and verifies that all other rules in the tuple are present in the policy. An example is shown in Figure 10. The tuple contains three rules with placeholder arguments. If a rule is found matching the first rule in the tuple, the arguments are extracted and substituted in the remaining rules; each of these rules must then be found in the policy.

Debug rules: The plugin detects rules containing debug types as either the domain or the type. Debug types can be specified by an analyst in the plugin configuration file.

Ineffective permissions: The plugin detects rules which grant some particular permission on a type, but do not grant some other particular permission on that type or some additional permissions on some other (related) type. All three sets of permissions can be specified by an analyst in the configuration file. An example is shown in Figure 11. If any permissions from the first set are granted on a file, then either all the permissions in the second set must be granted on the file, or the permissions in the third set must be granted on the file descriptor. The first rule grants `read` and `write` from the first set, and does not grant `open` from the second set; however, the second rule grants `use` on the file descriptor. The constraint is therefore satisfied.

Limitations The plugin allows an analyst to express very fine-grained information: this results in a somewhat complex configuration file.

If found:
```
file { write read append ioctl}
```
Look for either:
```
file { open }
```
or:
```
fd {use}
```
Rules:
```
allow a b:file { read write };
allow a b:fd use;
```

Figure 11: An example of the "ineffective permissions" functionality of the `unnecessary_rules` plugin.

4.7 Plugin 5: User `neverallows`

Goal `neverallow` rules can be used to specify permissions never to be granted in the policy. For example, Google uses `neverallow` rules extensively to prevent OEMs from circumventing core security structures of the policy. However, `neverallow` rules are only enforced at compile time in the normal SEAndroid policy development workflow: this means that a policy change may be committed into an OEM's VCS, only to later find out that it infringes one or more `neverallow` rules and therefore breaks the compilation. The `user_neverallows plugin` allows an analyst to define an additional set of `neverallow` rules, and be able to check at any time if they are respected by the policy. This can be very useful for OEM policy maintainers who would like to immediately make sure that developers contributing small policy changes do not introduce any undesired rules. The plugin enforces a list of custom user-defined `neverallow` rules on a policy, reporting any infringing rule.

Implementation The plugin checks each rule in the policy which matches any user-specified `neverallow`, and verifies that it does not grant any permission explictly forbidden in the `neverallow`. Custom `neverallow` rules can be defined by the analyst in the plugin configuration file, in the same syntax as they would be written in the policy.

Limitations The `user_neverallows` plugin processes each user-provided `neverallow` rule individually: therefore, it works best with small numbers of rules (tens of thousands).

5 EVALUATION

In order to show that SELint fulfills the requirements stated in Section 4.1, we solicited feedback from SEAndroid experts about their experience with SELint, as well as measured the tool's performance.

5.1 Expert Survey

Following Requirement 2, SELint is designed to be configured by an SEAndroid expert before regular developers can use it in their work flow. SEAndroid experts are, therefore, the main target audience of SELint. Developers are just expected to run SELint and verify that it doesn't produce new warnings on their policy modifications. Thus, in order to evaluate the usability and usefulness of SELint, we need to collect feedback from SEAndroid experts.

Materials In order to collect expert feedback about SELint, we prepared an evaluation questionnaire[3]. SELint itself was available for download via our public Github repository[4].

Procedure When collecting feedback on SELint, we wanted to focus on people that already have strong prior experience with SEAndroid policies. This choice is based on the fact that these experts are able to evaluate not just the tool itself, but also the default configuration we provide for its plugins. In order to obtain such feedback, we announced the SELint tool on the SEAndroid public mailing list[5]. This mailing list is a common forum where discussions among SEAndroid experts take place. We asked people to fill in the questionnaire after trying to use the tool on their Android tree.

Participants Three experts from three different companies evaluated SELint. Each had more than 2 years of experience with SEAndroid policies.

Results All respondents ranked SELint as easy to use, and its results as easy to interpret. They also agreed that functionality offered by SELint is not currently provided by any existing tools; they ranked SELint as being "*valuable*" for them for their current work on SEAndroid. Our free-form questions on the overall SELint experience gathered answers such as:

> "*I was able to use the tool to find things I wanted to fix with respect to over-privileged domains and useless rules.*"
>
> "*I think this just adds to the list of useful tools in policy development. The output is more user friendly than sepolicy-analyze and hopefully would appeal to those who only write policy infrequently - such as most OEMs.*"

Out of all the default plugins we provided with SELint, the `risky_rules` plugin caught the most attention and received the most positive feedback. This is as expected, given that this is the plugin that helps the most to directly evaluate the security of a SEAndroid policy. Plugins dealing with M4 macros were

[3] `goo.gl/forms/j9oUBL2wnEjOvpLs2`

[4] `github.com/seandroid-analytics/selint`

[5] `seandroid-list@tycho.nsa.gov`

also found to be useful, with respondents reporting that they actually adopted most or all suggestions for `global_macros` or `te_macros` in their SEAndroid policy. The `neverallow_rules` plugin got an expected answer to the question "*Do you plan to use the neverallow_rules plugin?*":

> "*Yes, to add rules I don't want in the policy, but where I don't want to add an actual neverallow. Neverallows end up in CTS, so you don't want to use them too much. As for OEM policy additions, sometimes neverallows are too strict and we just want to see what the linter picks up.*"

This is exactly the usage we envisioned for it: an ability for OEMs to enforce custom `neverallow` rules without them being checked by Android Compatibility Test Suite (CTS). Respondents also had some good points for future enhancements, such as implementing an easier setup wizard and automatically prompting to input the scores for types or permissions which do not have one in the `risky_rules` plugin.

Limitations In order to perform a better evaluation of SELint, we need a more extensive study with many more OEM developers who need to modify SEAndroid policies. However, this is difficult to achieve because of the following reasons. In order to try SELint, participants need to have their own custom Android tree and their own custom SEAndroid policies, since the tool targets OEM SEAndroid policy writers; this naturally limits the number of participants. In addition, people that actually have their own custom policy are usually engineers working for OEMs. They might not want to take part in our study because of corporate confidentiality concerns. Another difficulty is in setting up SELint, as one of our respondents noted. This is due to the fact that SELint relies on the policy representation library from SETools (Tresys, 2016) to perform policy parsing, and older versions of this library do not support some new SEAndroid policy elements, such as `xperms`. This, together with some compatibility issues between SEAndroid policy versions and SETools, made it harder for some users to setup the tool initially.

Despite these limitations, we believe the user feedback we received confirms that our goals and assumptions for SELint and the default configurations of its plugins are correct. In addition, this feedback gives us directions for future work discussed in Section 6. We also hope that we will receive more user feedback on our tool with time.

5.2 Performance Evaluation

In order to evaluate the performance of SELint we conducted a set of measurements, collecting execution time and memory usage. We consider these numbers to be the most important indicators for SELint, since it can be used either manually by a single person or automatically as part of a Continuous Integration (CI) process. The measurements were conducted on an off-the-shelf laptop with an Intel Core i7-4770HQ 2.20GHz CPU and 16GB of 1600MHz DDR3 RAM. Each measurement was repeated 10 times, and the average and standard deviation are presented in Table 4 and Table 5. The first table presents data for a public Intel tree, Android 5.1[6], and the second one for the public AOSP tree, master branch[7]. For all measurements we have measured the SELint core and each of its plugins separately. The big difference in performance between these two trees comes from the number of expanded rules in the source policies: for the Intel tree it is 99532, while for the AOSP tree it is 3081233. The execution time of the `unnecessary_rules` plugin scales differently than others, requiring almost the same time as the SELint core on the Intel tree and 10 times more than the SELint core on the AOSP tree. This is due not only to the different total number of rules in the two trees, but also to the number of rules that each domain has, since the plugin needs to check for ineffective rule combinations or permissions (see Section 4.6). The `parametrized macro` plugin is the only plugin that takes a considerable amount of time to run, especially on the AOSP tree. As explained in Section 4.4, this is due to the fact that we are currently not implementing any heuristics in our solution to the problem, and are just relying on exploration of the solution space. As a result, the current plugin should not be included into the default set of plugins executing au-

Component	Avg time (s)	Avg mem (MB)
SELint core	0.40 ± 0.01	99.53 ± 0.06
user_neverallows	0.43 ± 0.01	99.51 ± 0.05
simple macros	0.59 ± 0.02	99.94 ± 0.04
unnecessary_rules	0.65 ± 0.01	99.52 ± 0.08
risky_rules	1.06 ± 0.01	99.51 ± 0.05
parametrized macros	168.42 ± 2.17	446.52 ± 0.07

Table 4: Performance measurements for SELint on Intel Android tree with 99532 expanded rules

Component	Avg time (s)	Avg mem (MB)
SELint core	1.88 ± 0.02	212.11 ± 0.07
user_neverallows	1.89 ± 0.02	212.09 ± 0.07
simple macros	2.18 ± 0.03	219.03 ± 0.09
unnecessary_rules	20.25 ± 0.17	212.07 ± 0.06
risky_rules	3.23 ± 0.03	212.07 ± 0.07
parametrized macros	3210.03 ± 48.13	6031.84 ± 0.59

Table 5: Performance measurements for SELint on AOSP tree with 3081233 expanded rules

[6] `github.com/android-ia`

[7] `android.googlesource.com`

tomatically as part of a CI process, but should be used manually by an expert. The execution time and memory usage of the other plugins fit the desired use cases: given that normally an AOSP build takes at least half an hour to complete in a powerful CI infrastructure, an overhead of minutes and hundreds of MB of memory is considered acceptable.

6 DISCUSSION

While our evaluation showed that SELint is considered a valuable tool for analyzing SEAndroid policies, there are many areas for future work and improvements. The initial setup of SELint would benefit from an interactive procedure, allowing users to automatically detect and solve the possible mismatches between the installed libraries and policy versions. The `parametrized macro` plugin could provide an implementation based on a heuristic solution for the knapsack problem allowing users to obtain a partial solution, in order to save time and enable this plugin to be run as part of a CI infrastructure. More work is needed in order to polish the default configuration offered by the `risky_rules` plugin, and to provide a way for OEMs to easily, and maybe interactively, add scores for their own domains and types. We also need to conduct a study on how easy it is for SEAndroid experts to write new SELint plugins. Another future research direction is to investigate the possibility of using SELint together with a policy decompiler, in order to analyze OEM policies from available Android devices. This would provide additional input for SELint evaluation.

We continue to gather feedback from SELint users and SEAndroid experts to adjust SELint to their needs and requirements. Since SELint is open source software, and builds on existing official SEAndroid tools, we are planning to work with Google to include SELint in the set of SEAndroid tools provided with the AOSP tree.

REFERENCES

Amthor, P., Kuhnhauser, W., and Polck, A. (2011). Model-based safety analysis of SELinux security policies. In *NSS*, pages 208–215. IEEE.

Badger, L., Sterne, D., Sherman, D., Walker, K., , et al. (1995). Practical domain and type enforcement for UNIX. In *Security and Privacy*, pages 66–77. IEEE.

Bauer, M. (2006). Paranoid penguin: an introduction to Novell AppArmor. *Linux Journal*, (148):13.

Chu, P. C. and Beasley, J. E. (1998). A genetic algorithm for the multidimensional knapsack problem. *J heuristics*, 4(1):63–86.

Clemente, P., Kaba, B., et al. (2012). Sptrack: Visual analysis of information flows within selinux policies and attack logs. In *AMT*, pages 596–605. Springer.

Fowler, M. (2010). *Domain-specific languages*. Pearson Education.

Guttman, J. D., Herzog, A. L., Ramsdell, J. D., and Skorupka, C. W. (2005). Verifying information flow goals in security-enhanced Linux. *JCS*, 13(1):115–134.

Hanafi, S. and Freville, A. (1998). An efficient tabu search approach for the 0–1 multidimensional knapsack problem. *EJOR*, 106(2):659–675.

Harrison, M. A., Ruzzo, W. L., and Ullman, J. D. (1976). Protection in Operating Systems. *CACM*, 19(8).

Hurd, J., Carlsson, M., Finne, S., Letner, B., Stanley, J., and White, P. (2009). Policy DSL: High-level Specifications of Information Flows for Security Policies.

Jaeger, T., Sailer, R., and Zhang, X. (2003). Analyzing integrity protection in the SELinux example policy. In *USENIX Security*, page 5.

Kellerer, H., Pferschy, U., and Pisinger, D. (2004). *Knapsack problems*. Springer, Berlin.

Magazine, M. J. and Chern, M.-S. (1984). A note on approximation schemes for multidimensional knapsack problems. *MOR*, 9(2):244–247.

Marouf, S. and Shehab, M. (2011). SEGrapher: Visualization-based SELinux policy analysis. In *SAFECONFIG*, pages 1–8. IEEE.

Reshetova, E., Bonazzi, F., Nyman, T., Borgaonkar, R., and Asokan, N. (2016). Characterizing SEAndroid Policies in the Wild. In *ICISSP*.

Schaufler, C. (2008). Smack in embedded computing. In *Ottawa Linux Symposium*.

SELinux (2014). Userspace tools. `github.com/SELinuxProject/selinux`. Accessed: 29/09/15.

Smalley, S. and Craig, R. (2013). Security Enhanced (SE) Android: Bringing flexible MAC to Android. In *NDSS*, volume 310, pages 20–38.

Smalley, S., Vance, C., and Salamon, W. (2001). Implementing SELinux as a Linux security module. *NAI Labs Report*, 1(43):139.

Sniffen, B. T., Harris, D. R., and Ramsdell, J. D. (2006). Guided policy generation for application authors. In *SELinux Symposium*.

Tresys (2016). SETools project page. `github.com/TresysTechnology/setools`. Accessed: 18/05/16.

Wang, R. (2016). Automatic Generation, Refinement and Analysis of Security Policies. *`repository.lib.ncsu.edu/handle/1840.16/11139`*.

Wang, R., Enck, W., Reeves, D., et al. (2015). EASE-Android: Automatic Policy Analysis and Refinement for Security Enhanced Android via Large-Scale Semi-Supervised Learning. In *USENIX Security*.

Zhou, Y. and Jiang, X. (2012). Dissecting android malware: Characterization and evolution. In *Security and Privacy*, pages 95–109. IEEE.

Publication IV

Reshetova, Elena and Karhunen, Janne and Nyman, Thomas and Asokan, N. Security of OS-Level Virtualization Technologies. In *Nordic Conference on Secure IT Systems*, Location, pages, and other detailed information, October 2014.

Security of OS-Level Virtualization Technologies

Article · July 2014

DOI: 10.1007/978-3-319-11599-3_5 · Source: arXiv

4 authors, including:

Elena Reshetova
Aalto University
10 PUBLICATIONS **51** CITATIONS
SEE PROFILE

Thomas Nyman
Aalto University
13 PUBLICATIONS **37** CITATIONS
SEE PROFILE

N. Asokan
Aalto University
191 PUBLICATIONS **6,142** CITATIONS
SEE PROFILE

Security of OS-level virtualization technologies

Elena Reshetova[1], Janne Karhunen[2], Thomas Nyman[3], N. Asokan[4]

[1] Intel OTC, Finland
[2] Ericsson, Finland
[3] University of Helsinki, Finland
[4] Aalto University and University of Helsinki, Finland

Abstract. The need for flexible, low-overhead virtualization is evident on many fronts ranging from high-density cloud servers to mobile devices. During the past decade *OS-level virtualization* has emerged as a new, efficient approach for virtualization, with implementations in multiple different Unix-based systems. Despite its popularity, there has been no systematic study of OS-level virtualization from the point of view of security. In this report, we conduct a comparative study of several OS-level virtualization systems, discuss their security and identify some gaps in current solutions.

1 Introduction

During the past couple of decades the use of different virtualization technologies has been on a steady rise. Since IBM CP-40 [19], the first virtual machine prototype in 1966, many different types of virtualization and their uses have been actively explored both by the research community and by the industry. A relatively recent approach, which is becoming increasingly popular due to its light-weight nature, is *Operating System-Level Virtualization*, where a number of distinct user space instances, often referred to as *containers*, are run on top of a shared operating system kernel. A fundamental difference between OS-level virtualization and more established competitors, such as Xen hypervisor [24], VMWare [48] and Linux *Kernel Virtual Machine* [29] (KVM), is that in OS-level virtualization, the virtualized artifacts are global kernel resources, as opposed to hardware. This allows multiple virtual environments to share a common host kernel and utilize underlying OS interfaces. As a result, OS-level virtualization incurs less CPU, memory and networking overhead, which is important not only for *High Performance Computing* (HPC), such as dense cloud configurations, but also for resource constrained environments such as mobile and embedded devices. The main disadvantage of OS-level virtualization is that each container can only contain a system of the same type as the host environment, e.g. Linux guests on a Linux host.

An important factor to take into account in the evaluation of the effectiveness of any virtualization technology is the level of *isolation* it provides. In the context of OS-level virtualization isolation can be defined as separation between containers, as well as the separation between containers and the host. In order

to systematically compare the level of isolation provided by different OS-level virtualization solutions, one first needs to establish a common system model.

The goal of this study is to propose a generic model for a typical OS-level virtualization setup, identify its security requirements, and compare a selection of OS-level virtualization solutions with respect to this model. While other technologies as HW supported secure storage, various encryption primitives and specific CPU/memory features can enhance the security of OS-level virtualization solutions, they are left out of the scope of this paper and present the potential future work. To the best of our knowledge this is the first study of this kind that focuses on the security aspects of OS-level virtualization technologies. We base our analysis on information collected from the documentation and/or wherever possible the source code of the respective systems. As a result of this comparison section 9 identifies a number of gaps in the current implementation of Linux OS-level virtualization solutions.

2 Usage Scenarios

We identify the following common usage scenarios as motivation for OS-level virtualization in general. The first three originate from use cases in the context of warehouse scale computing. The latter two stem from security needs.

In **Server consolidation**, a set of distinct physical servers are substituted with a single physical server running a number of distinct virtual environments. Solutions based on hardware virtualization often require that the guest OS be modified; either to support the virtualization solution itself (as in the case of paravirtualization) or to facilitate interaction between the guest and host OSs by installing special-purpose components into the guest (as in full virtualization solutions such as VMWare, Virtual Box etc.) [47]. In contrast, one of the goals for OS-level virtualization is to provide a set of tools integrated into the OS to allow the creation and management of virtual environment without modifications to the software components placed inside a container. In *Virtual Private Server* (VPS) and cloud computing environments, where service providers grant superuser-level access in the rented virtual environments to customers, strict isolation between environments of different customers and the hosting provider is important unlike in server consolidation where all virtual environments are managed by the same entity.

Resource and application state management emerged from the need to run a number of distinct applications or multiple instances of a single application which require access to the same resources on a single machine, e.g. binding to the same network port. In addition by placing an application into a self-contained compartment it is possible to provide *Checkpoint and Restart* (CR) functionality [17,47,30]. CR allows processes to be moved between different physical or virtual environments. This can be useful for load-balancing or in high-availability environments, as well as software development and testing on different UNIX platforms.

A **Multi-OS experience** allows end-users the ability to use applications and services from different operating systems on the same device by the means of virtualization technology. While the OS-level virtualization is limited to systems sharing a common kernel, it can provide a way for the user to run a number of different OS variants on the same system. Since there are many new mobile operating systems on the rise such as Android, Tizen, FirefoxOS and the like, feature is likely to be useful for many experienced users. The need to share certain data, like the user's contacts or calendar, across the different OSs installed on the same device brings in an additional challenge for this use case.

Application or service isolation places critical and externally exposed services into separate sandbox environments that are able to contain damage in case sandboxed services become compromised. Sandboxing also makes it possible to delegate the administration of these services to third, possibly less trusted, parties [28].

The **Bring Your Own Device** (BYOD) policy [37] allows one physical device to be used simultaneously for personal and business needs resulting in a need of rigid separation between these two environments in order to guarantee user privacy while conforming to enterprise policies. Presenting separate environments to the end-user can also improve the usability of the solution [2], compared to domain separation by means of access control mechanisms alone.

3 System model

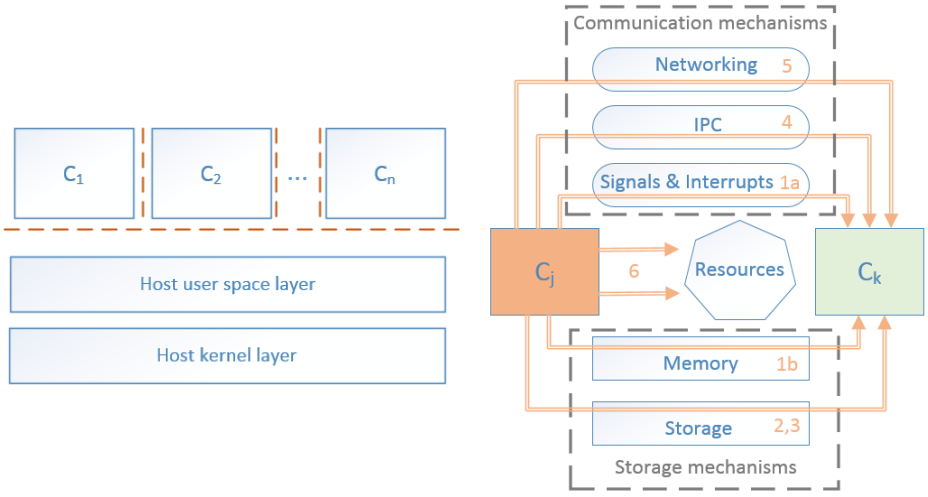
In Figure 1(a) we present a system model for a typical container setup that can support the types of usage scenarios we discussed in Section 2. There are a number of containers $C_1 \ldots C_n$ that run on a single physical host machine. The OS kernel is shared among all the containers, but the extent of shared host user space depends on a concrete setup (see Table 1):

Full OS installation & management corresponds to the most common case when the host user space layer comprises a complete OS installation with the container management layer on top. In this case some host resources may be shared between the host and one or more containers via bind-mounts [25] or overlay filesystems [20]. Each container can be one of two types:

- *Application containers* have a single application or service instance running inside. They are commonly used for application isolation or resource management referred to in Section 2.
- *System containers* have an entire OS user space installation and are commonly used for server consolidation.

Lightweight management corresponds to the case where the host user space layer consists of merely a light-weight management layer used to initialize and run containers. This setup can be argued to be more secure, as it exhibits a reduced attack surface compared to a complete underlying host system. Again, each container can be one of two types:

– *Direct application/service setup* refers to the case when only a single application or service is installed in the container. It is more suitable for application isolation scenarios in which, for instance, a banking application is run in a separate container isolated from the rest of a less trusted OS running in another container.
– *Direct OS setup* refers to the case when a container runs an entire OS user space installation. It can provide an end-user the appearance of simultaneously running multiple OS instances, and is therefore well suited for Multi-OS and BYOD environments.



(a) System model (b) Attacker model

Fig. 1. OS-level virtualization

Host user space layer	**Container**	
	Application/Service	*Full OS installation*
Full OS installation & management	Application container	System container
Lightweight management	Direct Application/Service setup	Direct OS setup

Table 1. Types of OS virtualization setups

The system model described above intentionally omits cases where containers $C_1 \ldots C_n$ are not independent, but arranged in a hierarchical structure. While some systems, such as FreeBSD jails [28], allow such setups, they are rarely used in practice and are therefore left beyond the scope of this report.

4 Attacker model and security requirements

The attacker is assumed to have full control over a certain subset $\bar{C}$ of containers. The remaining set C is assumed to be in the control of legitimate users. The goals of the attacker can be classified as follows:

- **Container compromise:** compromise $C_k \in C$ by means of illegitimate data access, *Man-in-the-Middle* (MitM) attacks or by affecting the control flow of instructions executed in $C_k \in C$.
- **Denial of Service:** disturb normal operation of the host or $C_k \in C$.
- **Privilege escalation:** obtain a privilege not originally granted to a $C_j \in \bar{C}$.

The above goals can be achieved via different types of attacks that can be roughly classified into distinct groups based on the interfaces available in a typical Single UNIX Specification compliant OS [45]. These attack groups can be further arranged into two classes based on the type of underlying mechanism: *attacks via communication mechanisms* and *attacks via storage mechanisms* (see Figure 1(b)). From this classification we derive a set of security requirements that each OS-level virtualization solution needs to fulfill. In the description below, numbers in parenthesis refer to arrows in Figure 1(b).

Separation of processes is a fundamental requirement that aims to isolate processes running in distinct containers to prevent $C_j \in \bar{C}$ from influencing $C_k \in C$ using interfaces provided by the operating system for process management, such as signals and interrupts (1a). In addition, it might be possible to directly access the memory of a process running in $C_k \in C$ by using special system calls, e.g. the *ptrace()* system call allows a debugger process to attach and monitor the memory of a debugged process (1b).

Filesystem isolation is required in order to prevent illegitimate access to filesystem objects belonging to $C_k \in C$ or the host (2).

Device isolation should protect device drivers shared between different containers and a host. Such drivers present another significant attack vector because they expose interfaces (3) to code running into the kernel space, which may be abused to gain illegitimate data access, escalate privileges or mount other attacks.

IPC isolation is needed in order to prevent $C_j \in \bar{C}$ from accessing or modifying data belonging to $C_k \in C$ being transmitted over different IPC channels (4). Such channels include traditional System V IPC primitives, such as semaphores, shared memory and message queues as well as POSIX message queues.

Network isolation aims to prevent attacks by $C_j \in \bar{C}$ via available network interfaces (5). In particular, an attacker can attempt to eavesdrop on or modify network traffic of the host or $C_j \in \bar{C}$, perform MitM attacks etc.

Resource management provides a way to limit the amount of resources available to each container depending on the system load. This is needed in order to prevent an attacker from exhausting physical resources available on a device, such as disk space or disk I/O limits, CPU cycles, network bandwidth and memory (6).

5 Terminology

We define the following terms that are going to be used through the remaining of this report and that might be not familiar for a reader without a Unix/Linux background:

- **Kernel space** refers to the part of the virtual memory that is used to run the OS kernel code, modules, drivers and extensions.
- **User space** denotes the part of the virtual memory that includes the user applications, system processes (daemons) and services.
- **Kernel resource** is a kernel structure referring to shared physical or virtual devices, system resources, such as memory or cpu time, or a set of identifiers used through the kernel.
- **Superuser/privileged user** is a special user in UNIX-like systems that is allowed to perform privileged system operations. UNIX classical *root* user is an example of such user.
- **Linux capabilities** is a set of predefined capabilities implemented in the Linux kernel for performing different privileged operations, such as mounting a filesystem or overwriting the system security policies. Such capabilities are often referred as POSIX capabilities.
- **Filesystem root** denotes the top-most directory in the UNIX filesystem hierarchy as it is visible to running processes.
- **Upstream/mainline Linux kernel** refers to the Linux kernel source code tree maintained at *kernel.org*. This is the official Linux kernel source that contains all the released and upcoming features.

6 Overview of technologies

6.1 FreeBSD Jails

The pioneering notion of Jails was first introduced in FreeBSD 4.0 in 2000 [28]. The motivation behind Jails was the need to have separate virtual compartments on a single host, combined with the ability to delegate a subset of the traditional superuser privileges to the root user for each compartment. A number of changes were introduced to the FreeBSD kernel in order to implement Jails. These include the hardening of the *chroot(2)* system call, basic isolation that would restrict a jailed process from communicating with processes outside the Jail, the ability to limit the visibility of processes via the *procfs* pseudo filesystem and *sysctl* interfaces, restrictions on TCP/IP networking, Jail-aware device drivers, and

the ability to restrict a root user inside a Jail from performing certain system calls. Later on the ability to have multiple IP-addresses per Jail, more powerful Jail management facilities and the ability to create hierarchical Jails were added.

6.2 Linux-VServer

The need for a mechanism similar to FreeBSD Jails in Linux led to the Linux-VServer project [8]. The first official release of the Linux-VServer occurred in 2003. In Linux-VServer, separate virtual environments are referred to as Virtual Private Servers (VPSs). They can be managed with the help of user space tools provided by the *util-vserver* package. Each VPS has its own context that contains all the information regarding the VPS; its name, allowed limits, bounded capabilities, scheduler information etc. In addition, the behavior of each VPS can be further adjusted by specifying context capabilities and flags that allow a VPS to modify its host name or hide network interfaces that a certain VPS is not permitted to access. The biggest downside with regards to deployability is the need to apply the Linux-VServer patches to kernel source code and recompile the kernel as the Linux-VServer changes are currently not integrated into the mainline Linux kernel development branch.

6.3 Solaris Zones

Solaris Zones/Containers project [35] was started in 2004 in order to provide a commercial OS-level virtualization solution. Sun engineers analyzed FreeBSD Jails and Linux-VServer solutions available at the time and concluded, that while the goals of the projects are similar, the depth of OS integration, quality of administrative tools and overall maturity of the aforementioned projects were not at a level needed to support commercial solutions. In addition, they also wanted to create a set of usable zone management tools that would enable the delegation of zone setup and configuration whenever possible to the administrators of a zone. The isolation provided by Solaris Zones is based on attaching a zone identifier to a process, and using it to restrict the visibility of the process across zone boundaries. The zone identifier is also used to determine process privileges inside non-global zones, System V IPC communication etc. Resource management is implemented using standard mechanisms provided in Solaris, such as entitlements, limits and partitions. When used together, they are able to ensure a minimal level of service, bound resource consumption and even the dedication of certain resources only to specific zones.

6.4 OpenVZ

The OpenVZ project [11] is another open source OS-level virtualization solution for Linux begun in 2005. It is currently part of the commercial Parallels Cloud Server solution [12]. OpenVZ uses the term Virtual Environments (VEs) [47] to refer to containers. The implementation consists of set of kernel changes and

user space tools. The OpenVZ kernel is based on the Red Hat Enterprise Linux kernel, which in turn is based on the relatively old 2.6.32 upstream Linux kernel. However, OpenVZ developers have integrated many of their kernel modifications into the upstream kernel. Hence, the project's main user space tool, *vzctl*, can be used with both upstream and OpenVZ kernels, but the developers "recommend using the OpenVZ kernel for security, stability and features". The OpenVZ project was also the first one to implement the *Checkpoint and Restart* (CR) functionality for for VEs [30]. CR allows processes to be moved between different physical or virtual environments. This can be useful for load-balancing or in high-availability environments, as well as software development and testing on different UNIX platforms.

6.5 LxC

The Linux Containers (LxC) project [9] is the only currently available OS-level virtualization solution for Linux that consists only of a set of user space tools. This is possible because LxC utilizes only those virtualization features integrated into the upstream Linux kernel. This gives LxC an advantage over other Linux-based projects, because the process of applying patches and compiling a specific version of the Linux kernel can become a non-trivial task even for experienced Linux users. Another differentiating feature of LxC is the ability to use Linux Security Modules (LSMs) [41] to harden a container setup. Apparmor [1] and SELinux [38] profiles are officially supported, but in principle, other existing LSMs, such as Smack [15] could be used as well.

6.6 Cells/Cellrox

The Cells architecture [22] and the commercial Cellrox solution [2] built on the Cells architecture are the only open source OS-level virtualization solutions for smartphones. The primary design goal behind Cells is to support the *Bring Your Own Device* (BYOD) policy [37] on the Android platform. BYOD allows one physical device to be used simultaneously for personal and business needs resulting in a need of rigid separation between these two environments in order to guarantee user privacy while conforming to enterprise policies. Cells allows a user to have two or more virtual phones on the single Android device, e.g. one for personal use and another, which can be controlled by the user's employer and can contain confidential company data and applications. The user can switch between the virtual phones using a special icon on the Android home screen. Similarly to LxC, Cells utilizes upstream kernel features to isolate virtual phones. However, since Android has some non-standard Linux extensions, the developers of Cells had to implement a number of additional isolation mechanisms. The primary example are the changes done to the Binder driver in order to support IPC isolation on Android.

	container structure	**separate namespaces**
pros	simplicity, convenience	flexibility, incremental introduction of containerization
cons	possible information duplication, less flexibility	increased complexity
used by	FreeBSD, SolarisZones, Linux-VServer, OpenVZ	Linux-VServer, OpenVZ, LxC, Cells

Table 2. Comparison of containerization approaches

7 Comparison

Following [17], we define the notion of a *kernel namespace* as a set of identifiers representing a class of global kernel resources, such as process and user ids, IPC objects or filesystem mounts. The OS-level virtualization in the upstream Linux kernel is based on the usage of different kernel namespaces. The subsections below introduce the relevant namespaces and compare selected OS-level virtualization solutions, highlighted in bold in the previous paragraph, based on the security requirements listed in section 4.

7.1 Separation of processes

The primary isolation mechanism required from any OS-level virtualization solution is that it is able to distinguish processes running in different containers from those running on the host, limit cross-container process visibility and to prevent memory and signaling-level attacks described in the section 4. The simplest solution to this problem is to embed a container identifier C_i into the process data structure and to check the scope and the permissions of all syscall invocations.

FreeBSD Jails, Solaris Zones, OpenVZ and Linux-VServer implementations follow this approach by linking a structure describing the container to the process data structure. However, unlike FreeBSD and Solaris, the data structures describing OpenVZ and Linux-VServer containers are not used to achieve process separation. They only store related container data such as resource limits and capabilities. Instead, OpenVZ, Linux-VServer, LxC and Cells use *process id (pid) namespaces* that are part of the mainline Linux kernel. A pid namespace is a mechanism to group processes in order to control their ability to see (for example via *proc* pseudo-filesystem) and interact (for example by sending signals) with one another. The pid namespaces also provide pid virtualization: two processes in different pid namespaces may have the same pid.

Having a separate structure describing a container and storing a pointer to it in the process task structure is a convenient way to have all the relevant information concerning the container in one place. However, the upstream Linux kernel has followed a different approach of grouping different kernel resources into separate namespaces and using these namespaces to build containers. This approach incurs additional complexity, but adds the flexibility to choose a combination of

namespaces that best fits the desired use case. It also allows gradual introduction of namespaces to an existing system, like the upstream Linux kernel, which also helps in testing and verification of the implementation [17]. Furthermore, it avoids information duplication when both the process and the container structures have similar information. The pros and cons of these two approaches are summarized in Table 2.

In addition to the ability to isolate and virtualize process ids, the upstream Linux kernel also allows virtualization and isolation of the user and group identifiers with the help of *user namespaces*. Typically the root user has all the privileges to perform various system administration tasks and is able to override all access control restrictions. However, it is not desired that a root user running inside a container would be given the privileges of the host root user. Therefore, the Linux user namespace implementation interprets a given Linux capability as authorizing an action within that namespace: for example, the *CAP_SYS_BOOT* capability inside a container grants the authority to reboot that container and not the host. Moreover, many capabilities such as *CAP_SYS_MODULE* cannot be safely granted for container in any meaningful manner. When a process attempts to perform an action guarded by such capability, the kernel always checks if the process possesses this capability in the host user namespace. All Linux OS-level virtualization solutions support the option of starting a new user namespace for each container, but all the related configuration such as mapping the user identifiers between the host and the container must be done manually.

7.2 Filesystem isolation

The filesystem is one of the most important OS interfaces that allows processes to store and share data as well as to interact with one another. In order to prevent filesystem-based attacks described in section 4, it should be possible to isolate the filesystem between containers and to minimize the sharing of the data. The amount of sharing needed between the host and each container depends on the usage scenario. In the case of application isolation, it is not worthwhile to completely duplicate the OS setup inside a container and therefore some parts of the filesystem, such as common libraries, need to be securely shared with the host. On the other hand in the case of server consolidation, quite often it is best to completely separate the filesystems and create container filesystems from scratch.

All Linux-based OS virtualization solutions utilize a *mount namespace* that allows separation of mounts between the containers and the host. The design of upstream Linux mount namespaces[17] has been influenced by private namespaces [34] in Plan 9 from Bell Labs [33]. Namespaces in Plan 9 are file-orientated, and the principal purpose is to facilitate the customization of the environment visible to users and processes. Since all Linux based systems create each container within a new mount namespace, all the internal mount events are only effective inside the given container. However, it is important to underline that the mount namespace by itself is not a security measure. Running a container in

a separate mount namespace does not give any additional guarantees concerning the data isolation between the containers since containers inherit the view of filesystem mounts from their parent and thus are able to access all parts of the filesystem similarly.

A typical approach for process filesystem access containment is by using the *chroot()* system call where process is bound within a subtree of the filesystem hierarchy. If desired, resources may be shared with the host by mounting them within the subtree visible inside the container. Since the *chroot()* system call [7] only affects pathname resolution, privileged processes (i.e. processes with the *CAP_SYS_CHROOT* privilege) can escape the chroot jail. This can be done for example by changing the root directory again via *chroot()* to a subdirectory relative to their current working directory. Of the virtualization solutions under comparison, only Cells relies on *chroot()* alone. Some systems, such as Linux-VServer utilize a *Secure chroot barrier* [8] to prevent processes in a VPS from escaping the modified environment.

Another approach, utilized by for instance LxC, is to not only modify the root directory for processes in a container, but modify the *root filesystem* as well. This can be achieved with the Linux specific *pivot_root()* system call [7], which is typically used during boot to change from a temporary root filesystem (e.g. an initrd) to the actual root filesystem. As its name suggests, the *pivot_root()* system call moves the mountpoint of the old root filesystem to a directory under the new root filesystem, and puts the new root filesystem at its place. When done inside a mount namespace, the old root filesystem can be unmounted, thus rendering the host root filesystem inaccessible for processes inside the container, without affecting processes belonging to the root mount namespace on the host system. At the time of writing, the implementation of *pivot_root()* also changes the root directory and current working directory of the process to the mountpoint of the new root filesystem if they point to the old root directory. OpenVZ relies on this behavior and uses the *pivot_root()* system call alone. However, as the behavior with regards to the current root directory and the current working directory remains unspecified, proper usage dictates that the caller of *pivot_root()* must ensure that processes with root directory or current working directory at the old root operate correctly regardless of the behavior of *pivot_root()*. To ensure this, LxC changes the root directory and current working directory to the mountpoint of the new root before invoking *pivot_root()*.

FreeBSD and Solaris also provide a sandbox-like environment for each jail/zone using similar *chroot()*-like calls that are claimed to avoid above mentioned security vulnerabilities [28], [35]. Mounting and unmounting of filesystems is prohibited by default for a process running inside a jail unless different *allow.mount.** options are specified.

A separate user namespace per container can further strengthen the filesystem isolation by mapping the user and group ids to a less privileged range of host uids and groups. Together with a mount namespace and a *pivot_root* environment it strengthens protection against filesystem-based attacks described in 4.

7.3 Device isolation

In Unix, device nodes are special files that provide an interface to the host device drivers. In classical Unix configurations, the device nodes are separated from the rest of the filesystem and their inodes are placed in the */dev* directory. In the case of Linux, this task is usually performed by the udevd daemon process issuing the *mknod* system call upon receiving the event from the kernel. Device nodes are security-sensitive since an improperly exposed or shared device inside a container can lead to a number of easy attacks (see section 4). In the simplest example, if a container has an access to */dev/kmem* and */dev/mem* nodes, it is able to read and write all the memory of the host. Thus, in order to isolate containers from one another it is important to prevent containers from creating new device nodes and to make sure that containers are only allowed to access a "safe" set of devices listed below:

1. **Purely virtual devices**, such as pseudo-terminals and virtual network interfaces. The security guarantee comes from the fact that these devices are explicitly created for each container and not shared.
2. **Stateless devices**, such as *random*, *null* and others. Sharing these devices among all containers and the host is safe because they are stateless.
3. **User namespace-aware devices**. If a device supports verifying process capabilities in the corresponding user namespace, then it is safe to expose such device to a container, because the specified limitations will be enforced. The current 3.14-rc2 upstream kernel does not have any physical devices supporting this feature, but they are expected to appear in the future.

All compared systems allow the system administrator to define a unique set of device nodes for each container and by default create only a small set of stateless and virtual devices. In Linux, creation of new device nodes within containers can be controlled by limiting access to the *CAP_SYS_MKNOD* Linux capability and by ensuring that all mountpoints inside containers have the *nodev* flag set.

The biggest difference of the Cells implementation is the addition of a "*device namespace*" that attempts to make the Linux input/output devices namespace-aware. Cells assumes the host to have a single set of input/output devices and multiplexes access to the physical host device via virtual devices created in each container. One virtual device at a time is allowed to access physical devices, based on whether an application from a given container is "on the foreground" (ie. visible on the screen) or not. Security-wise such an exclusive-access solution is comparable to the "purely virtual" devices category mentioned above and can be considered safe.

As mentioned above, Linux device drivers controlling physical devices are currently not namespace-aware and thus cannot be securely used inside containers. Quite commonly these devices assume only one controlling master host and require privileges that are hard to grant for a unprivileged container securely (unless the device is used exclusively by a single container). In other words, namespace support inside the device drivers would require extensive modifications to the existing driver code base.

	Layer 3 bind filtering	**Layer 3 VNI**	**Layer 2 VNI**
traffic shaping and policing	no	yes	yes
separate routing and filtering tables	no	no	yes
used by	FreeBSD Jails, Linux-VServer	Solaris Zones, OpenVZ	Solaris Zones, OpenVZ, LxC, Cells

Table 3. Comparison of network isolation

7.4 IPC isolation

In order to achieve IPC isolation between containers, processes must be restricted to communicate via certain IPC primitives only within their own container. If the filesystem isolation is done correctly (see section 7.2), then filesystem-based IPC mechanisms (such as UNIX domain sockets and named pipes) are automatically isolated because the processes are not able to access filesystem paths outside of their own container. However, the isolation of the rest of the IPC objects (such as System V IPC objects and POSIX message queues) requires additional mechanisms. In Linux these IPC objects are isolated with the help of the *IPC namespaces* that allow the creation of a completely disjoint set of IPC objects. Linux-VServer, OpenVZ, LxC and Cells all spawn a new IPC namespace for each container in order to achieve the required isolation.

In addition to using IPC namespaces, Cells also has to implement namespace support for the Binder system since it is the primary IPC mechanism on the Android OS. The solution [10] includes having a separate Context Manager for each IPC namespace that is able to resolve Binder addresses only in that namespace and therefore provide isolation of Binder addresses between different containers.

Solaris Zones follow a different approach to isolate IPC objects that are not filesystem path-based. A zone ID is attached to each object based on the zone ID of the process that creates it, and processes are not able to access objects from other zones. An exception is made only for an administrator in the global zone that can access and manage all the objects. FreeBSD simply blocks SysV IPC object-related system calls if such calls are issued from within a jail. The *allow.sysvipc* option allows SysV IPC mechanisms for jailed processes but lacks any isolation between jails.

7.5 Network isolation

The main goal of network isolation is to prevent network-based attacks described in section 4. Moreover, in order to fulfill the server consolidation and resource management use cases, it also needs to provide a virtualized view of the network stack.

Network isolation methods differ in terms of the OSI layer of the TCP/IP stack where the isolation is implemented (see Table 3 for a comparison between these implementations). FreeBSD and Linux-VServer implement network isolation on Layer 3 with the help of bind filtering. They restrict a *bind()* call made from within a container to a set of specified IP addresses and therefore processes are only allowed to send and receive packets to/from these addresses. The benefit of such an approach is the small amount of code that needs to be modified in the network implementation and a minimal performance overhead. However, the downside is that a lot of the standard networking functionality is not accessible for a process inside a container such as obtaining an address from the Dynamic Host configuration Protocol (DHCP), acting as a DHCP server or the usage of routing tables.

Another approach, supported by Solaris Zones and OpenVZ, provides a Layer 3 virtualized network interface (VNI) for each container. Compared to bind filtering this implementation is more flexible since it allows the configuration of different traffic control settings, such as traffic shaping and policing, from within the container. The Layer 3 implementation provided by OpenVZ is called *venet*, while Solaris uses the term *shared-IP zone*.

The third approach includes providing a Layer 2 virtualized network interface for each container with a valid Link layer address. This gives containers the ability to use many features that are not supported by the previous two solutions, such as DHCP autoconfiguration, separate routing information and filtering rules. This approach can also support a broader set of network configurations. However, the primary downsides include a performance penalty and the inability to control the container networking setup from the host. The latter can be important for the server consolidation case if the host administrator needs to be in the control of the overall network configuration. OpenVZ, Solaris, LxC and Cells all support the creation of the Layer 2 virtualized interfaces. On Linux platforms this feature is called virtual Ethernet (*veth*). On Solaris a similar configuration is named *exclusive-IP zone*.

The Linux Layer 2 network isolation is based on the concept of a *network namespace* that allows the creation of a number of networking stacks that appear to be completely independent. The simplest networking configuration for a container running in a separate network namespace includes a pair of virtually linked Ethernet (*veth*) interfaces and assigning one of them to the target namespace while keeping the other one in the host namespace. After the virtual link is established, interfaces can be configured and brought up [6].

Linux provides multiple ways for connecting containers to physical networks. One option is connecting the *veth* interface and the host physical interface by using a virtual network bridge device. Another option is to utilize routing tables to forward the traffic between virtual and physical interfaces. When a virtual bridge device is used, all container and host interfaces are attached to the same link layer bridge and thus receive all link layer traffic on the bridge. However, in the case of route configuration, containers are not able to communicate with each other unless a network route is explicitly provided. Also in the latter case,

	rlimits	***cgroups***
scope	per process, inheritable	per process group, inheritable
managed resources	memory(limited), CPU(limited), filesystem, number of threads	memory, CPU, block I/O, devices, traffic controller
action when limit is reached	resource request denial and process termination	resource request denial, possibility to have a custom action
used by	Linux-VServer, Cells	OpenVZ, LxC, Linux-VServer, Cells

Table 4. Comparison of Linux resource management mechanisms

container addresses are not visible to outsiders like in bridged mode. Another way of providing network connectivity for containers is to use the *MACVLAN* interface [9] that allows each container to have its own separate link layer address. MACVLAN can be set to operate in a number of modes. In a private mode containers cannot communicate with each other or the host making it the strictest isolation setup. The bridge mode allows containers to communicate with one another, but not with the host. The Virtual Ethernet Port Aggregator (*VEPA*) mode by default isolates containers from one another, but leaves the possibility to have an upstream switch that can be configured to forward packets back to the corresponding interface. Currently LxC is the only solution that can support all the *MACVLAN* modes.

7.6 Resource limiting

A good virtualization solution needs to provide support for limiting the amount of primary physical resources allocated to each container in order to prevent containers from carrying out denial of service attacks described in section 4.

Since the 9.0 release FreeBSD utilizes Hierarchical Resource Limits (RCTL) to provide resource limitation for users, processes or jails [13]. RCTL supports defining an action in case a specified limit is reached: deny new resource allocation, log a warning, send a signal (for example SIGHUP or SIGKILL) to a process that exceeded the limit or to send a notification to the device state change daemon.

Solaris implements resource management for zones using a number of techniques that can be either applied to a whole zone or to a specific process inside a zone. Resource partitioning, called *resource pools*, allows defining a set of resources, such as a physical processor set, to be exclusively used by a zone. A dynamic resource pool allows to adjusting the pool allocations based on the system load. Resource capping is able to limit the amount of the physical memory used by a zone.

The traditional way of managing resources on BSD-derived systems is the *rlimits* mechanism that allows specifying soft and hard limits for system resources for each process. Cells and Linux-VServer utilize *rlimits* to do resource

management for containers. However, the main problem of rlimits is that it does not allow specifying limits for a set of processes or to define an action when a limit is reached. Also the CPU and memory controls are very limited and do not allow specifying the relative share of CPU time, number of virtual pages resident in RAM or physical CPU or memory bank allocations.

In an attempt to address some of these limitations, OpenVZ and Linux-VServer have implemented custom resource management extensions, such as new limits for the maximum size of shared and anonymous memory or new CPU scheduler mechanisms. In addition both virtualization solutions added the possibility to specify resource limits per container.

Linux Control Groups (cgroups) [3] is a relatively new mechanism that aims to address the downsides of *rlimits*. It allows arranging a set of processes into hierarchical groups and performs resource management for the whole group. The CPU and memory controls provided by *cgroups* are rich, and in addition it is possible to implement a complex recovery management in case processes exceed their assigned limits. LxC, Linux-VServer, OpenVZ and Cells provide a way to use cgroups as a container resource management mechanism.

Table 4 presents a comparison of different aspects between *rlimits* and *cgroups*. A combined use of these mechanisms allows protecting the container from a set of DoS attacks directed towards the CPU, memory, disk I/O and filesystem (*rlimits* combined with *filesystem quotas*). However, the future direction is to aggregate all resource management to *cgroups*, and allow *rlimits* to be changed by a privileged user inside a container[5].

8 Related work

A number of previous studies have compared different aspects of the OS-level virtualization to other virtualization solutions. Padala et al. [32] analyze the performance of Xen vs. OpenVZ in the context of server consolidation. Chaudhary et al. [26], Regola et al. [36] and Xavier et al. [42] perform comparisons of different virtualization technologies for HPC. Yang et al. [43] study the impact of different virtualization technologies for the performance of the Hadoop framework [46].

The Capsicum sandboxing framework [39] introduced in FreeBSD 9 isolates processes from global kernel resources by disabling system calls which address resources via global namespaces. Instead, resources are accessed via capabilities which extend Unix file descriptors. Linux has a similar mechanism, called *seccomp* [18], that allows a process to restrict a set of systems calls that it can execute. Both Capsicum and *seccomp* require modifications to existing applications.

While there are OS-level virtualization solutions such as ICore [4] and Sandboxie [14] in existence for Microsoft Windows as add-on solutions, we have left

[5] documentation in source code of `http://lxr.linux.no/#linux+v3.13.5/kernel/sys.c#L1368`

	separation of processes	***file-system isolation***	***IPC isolation***	***device isolation***	***network isolation***	***resource limiting***
achieved by	pid ns	mount ns, pivot_root			network ns, veth, MACVLAN	rlimits, cgroups
	user ns		ipc ns	cgroups device controller, exclusive device usage		
open problems	security ns		IPC extensions	device ns, (pseudo)random devices, hotplug support	n/a	incomplete cgroups

Table 5. Summary of OS-level virtualization in upstream Linux kernel

them out this report's scope due to their closed nature. Authors are not aware of any OS-level virtualization solutions for Mac OS X or iOS.

In addition to the OS-level virtualization solutions under comparison in this study, researchers have developed a number of other technologies. An attempt by Banga et al. [23] to do fine-grained resource management led to the creation of a new facility for resource management in server systems called *Resource Containers*. Zap [31] allows the grouping of processes into *Process Domains* (PODs) that provide a virtualized view of the system and support for CR. An OS-level virtual machine architecture for Windows is proposed by Yu et al. [44]. A partial OS-level virtualization is provided by the PDS environment by Alpern et al. [21]. Wessel et al. [40] propose a solution for isolating user space instances on Android similar to the Cells/Cellrox. The solution by Wessel et al. has a special focus on security extensions, such as remote management, integrity protection and storage encryption.

9 Discussion and Conclusions

All compared systems implement core container separation features in terms of the memory, storage, network and process isolation. However, while the initial innovation around containers happened on FreeBSD and Solaris, the mainline Linux has caught up in terms of features and the flexibility of the implementation. Linux is likely to have a complete user space process environment virtualization in course of time. Given the scale of deployment of Linux and the maturity of its OS-level virtualization features, we focus on Linux in the rest of this section.

Table 5 summaries the state of the OS-level virtualization supported by the current upstream Linux kernel. The first row shows how each type of isolation discussed in section 7 can be achieved using the currently available techniques. The second row presents a number of gaps that are briefly described below.

Security namespaces. In order to reduce security exposure and adhere to the *principle of least privilege*, many OSs provide an integrated mandatory ac-

cess control (MAC) mechanism. MACs can be used to strengthen the isolation between different containers and the host, as well as to enforce MAC policies for processes inside containers. The latter is especially important when the container has a full OS installation, because it usually comes with pre-configured MAC policies. Therefore, OS-level virtualization solutions should support the ability to use the common MAC mechanisms in the underlying host kernel to enforce independently defined (container-specific) MAC policies. However, currently none of the compared solutions fulfills this requirement. Linux kernel developers plan to address this limitation in the future by introducing a *security namespace* that would make LSMs container-aware.

IPC extensions. While IPC namespaces and filesystem isolation techniques cover most of the inter-process communication methods available on Linux, exceptions exist. For example *Transparent Inter-process Communication* (TIPC) [16] is not currently covered. TIPC is a network protocol that is designed for an inter-cluster communication. Usage of such methods would break the IPC isolation borders between containers and if the given features are not needed, they should be disabled from the kernel configuration.

Device namespaces. As discussed in section 7.3, secure access to device drivers from within a container remains an open problem. One way to approach it would be to create a new namespace class (a *device namespace*) and group all devices to belong in their own device namespaces in hierarchical manner, following the generic namespace design pattern. Given this, only processes within the same device namespace would be allowed to access devices belonging in it. However, since the core of such functionality would resemble more access/resource control than a fully featured namespace, it was initially decided to implement the functionality as a separate *cgroups* device controller. The discussions defining the full notion of the device namespace and its functionality continue in the kernel community [5].

(Pseudo)random number generator devices. In section 7.3 we stated that using stateless devices such as */dev/random* or */dev/urandom* are secure within containers due to their stateless nature. This means that even if two containers share the same device, they cannot predict or influence the output from another device node within another container. However, it is important to note that exposing blocking devices, such as */dev/random*, poses a Denial-of-Service possibility. A malicious container can exhaust all available entropy and block the */dev/random* from being used in all other containers and the host, making it impossible to perform cryptographic operations requiring random input. Even if only non-blocking */dev/urandom* is exposed, there is a theoretical possibility that a malicious container can predict the random output for another container or a host. For example in [27] Dodis et al. give an assessment of both */dev/random* and */dev/urandom* showing that these devices do not accumulate entropy properly. A complete solution would be to implement a separate random device per namespace or even introduce a namespace for (pseudo)random number generators.

Hotplug support. Desktop Linux relies heavily on the dynamic nature of device nodes. Once new devices are plugged in to the system, the kernel generates an *uevent* structure notifying the user space of the new hardware. As briefly explained in section 7.3, *Uevent* is typically handled by the *udevd* daemon which configures the device for system use. Traditionally it has also created the corresponding device node after device setup. As far as containers are concerned, this setup is risky and complicated - containers should not be allowed to configure hardware and/or have permissions for creating the new device nodes. As a result, safe device hotplug for containers remains an open problem.

Incomplete implementation of *cgroups*. As was mentioned in the section 7.6, the current goal of the upstream Linux is to integrate all features supported by *rlimits* into the *cgroups* resource management. However this has not been done yet and currently remains as work in progress.

References

1. AppArmor project wiki. `http://wiki.apparmor.net/index.php/Main_Page`.
2. Cellrox project. `http://www.cellrox.com/`.
3. Cgroups. `https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt`.
4. iCore project page. `http://icoresoftware.com/`.
5. Linux Containers mailing list. `http://lists.linuxfoundation.org/pipermail/containers/2013-September/033466.html`.
6. Linux Network Namespaces. `http://www.opencloudblog.com/?p=42`.
7. Linux Programmer's Manual pages (release 3.35).
8. Linux-VServer project. `http://linux-vserver.org`.
9. LxC project. `http://linuxcontainers.org/`.
10. Namespace support for Android binder. `http://lwn.net/Articles/577957/`.
11. OpenVZ project. `http://openvz.org`.
12. Parallels products page. `http://www.parallels.com/products`.
13. RCTL. `https://wiki.freebsd.org/Hierarchical_Resource_Limits`.
14. Sandboxie project page. `http://www.sandboxie.com/`.
15. Smack project. `http://schaufler-ca.com/home`.
16. TIPC project. `http://tipc.sourceforge.net/`.
17. Biederman. Multiple Instances of the Global Linux Namespaces. In *Linux Symposium*, pages 101–112, 2006.
18. Corbet. Seccomp and sandboxing. `http://lwn.net/Articles/332974/`.
19. Creasy. The origin of the VM/370 time-sharing system. *IBM Journal of Research and Development*, pages 483–490, 1981.
20. Edge. Another union filesystem approach. `https://lwn.net/Articles/403012/`.
21. Alpern et al. PDS: a virtual execution environment for software deployment. In *VEE*, pages 175–185, 2005.
22. Andrus et al. Cells: a virtual mobile smartphone architecture. In *ACM SOSP*, pages 173–187, 2011.
23. Banga et al. Resource containers: A new facility for resource management in server systems. In *OSDI*, pages 45–58, 1999.
24. Barham et al. Xen and the art of virtualization. *ACM SIGOPS OSR*, pages 164–177, 2003.

25. Bhattiprolu et al. Virtual servers and checkpoint/restart in mainstream Linux. *ACM SIGOPS OSR*, pages 104–113, 2008.
26. Chaudhary et al. A comparison of virtualization technologies for HPC. In *AINA*, pages 861–868, 2008.
27. Dodis et al. Security analysis of pseudo-random number generators with input:/dev/random is not robust. In *2013 ACM SIGSAC*, pages 647–658, 2013.
28. Kamp et al. Jails: Confining the omnipotent root. In *SANE*, page 116, 2000.
29. Kivity et al. KVM: the Linux virtual machine monitor. In *Linux Symposium*, volume 1, pages 225–230, 2007.
30. Mirkin et al. Containers checkpointing and live migration. In *Linux Symposium*, pages 85–92, 2008.
31. Osman et al. The design and implementation of Zap: A system for migrating computing environments. *ACM SIGOPS OSR*, pages 361–376, 2002.
32. Padala et al. Performance evaluation of virtualization technologies for server consolidation. *HP Labs Tec. Report*, 2007.
33. Pike et al. Plan 9 from Bell Labs. In *UKUUG*, pages 1–9, 1990.
34. Pike et al. The Use of Name Spaces in Plan 9. In *5th workshop on ACM SIGOPS European workshop*, pages 1–5, 1992.
35. Price et al. Solaris Zones: Operating System Support for Consolidating Commercial Workloads. In *LISA*, pages 241–254, 2004.
36. Regola et al. Recommendations for virtualization technologies in high performance computing. In *IEEE CloudCom*, pages 409–416, 2010.
37. Shim et al. Bring Your Own Device (BYOD): Current Status, Issues, and Future Directions. 2013.
38. Smalley et al. Implementing SELinux as a Linux security module. *NAI Labs Report*, 1:43, 2001.
39. Watson et al. Capsicum: Practical Capabilities for UNIX. In *USENIX*, pages 29–46, 2010.
40. Wessel et al. Improving Mobile Device Security with Operating System-Level Virtualization. In *Security and Privacy Protection in Information Processing Systems*, pages 148–161. 2013.
41. Wright et al. Linux security module framework. In *Linux Symposium*, pages 604–617, 2002.
42. Xavier et al. Performance evaluation of container-based virtualization for high performance computing environments. In *PDP*, pages 233–240, 2013.
43. Yang et al. Impacts of Virtualization Technologies on Hadoop. In *ISDEA*, pages 846–849, 2013.
44. Yu et al. A feather-weight virtual machine for windows applications. In *VEE*, pages 24–34, 2006.
45. The Open Group. *The Single UNIX Specification: Authorized Guide to Version 4*. 2010. `http://www.unix.org/version4/theguide.html`.
46. Kizza. Virtualization Infrastructure and Related Security Issues. In *Guide to Computer Network Security*, pages 447–464. 2013.
47. Kolyshkin. Virtualization in Linux. *White paper, OpenVZ*, 2006.
48. Rosenblum. VMwares Virtual Platform. In *Hot Chips*, pages 185–196, 1999.

Publication V

Reshetova, Elena and Bonazzi, Filippo and Asokan, N. Randomization Can't Stop BPF JIT Spray. In *Network and System Security: 11th International Conference*, Location, pages, and other detailed information, August 2017.

Randomization Can't Stop BPF JIT Spray

Elena Reshetova[1], Filippo Bonazzi[2], and N.Asokan[2,3]

[1] Intel OTC, Espoo, Finland
[2] Aalto University, Helsinki, Finland
[3] University of Helsinki, Helsinki, Finland

Abstract. The Linux kernel Berkeley Packet Filter (BPF) and its Just-In-Time (JIT) compiler are actively used in various pieces of networking equipment where filtering speed is especially important. In 2012, the Linux BPF/JIT compiler was shown to be vulnerable to a JIT spray attack; fixes were quickly merged into the Linux kernel in order to stop the attack. In this paper we show two modifications of the original attack which still succeed on a modern 4.4 Linux kernel, and demonstrate that JIT spray is still a major problem for the Linux BPF/JIT compiler. This work helped to make the case for further and proper countermeasures to the attack, which have then been merged into the 4.7 Linux kernel.

Keywords: Network Security, Berkeley Packet Filter, JIT Spray

1 Introduction

Attackers seeking to compromise Linux systems increasingly focus their attention on the kernel rather than on userspace applications, especially in mobile and embedded devices. The primary reason for this change is the extensive work done over the years to limit the damage when a userspace application is exploited. For example, the latest releases of Android have SEAndroid policies that do not allow a compromised application to get any significant control over the OS itself. On the contrary, finding a vulnerability in the kernel almost always leads to compromise of the whole device.

Many kernel (and userspace) vulnerabilities are the result of programming mistakes, such as uninitialized variables, missing boundary checks, use-after-free situations etc. While it is important to develop tools to help finding these mistakes, it is impossible to fully avoid them. Moreover, even when a vulnerability is discovered and fixed in the upstream kernel, it takes approximately 5 years for the fix to be propagated to all end user devices [9].

The Kernel Self Protection Project (KSPP)[4] tries to eliminate whole classes of vulnerabilities that might lead to successful exploits, by implementing various hardening mechanisms inside the kernel itself. An important part of the project is to create Proof Of Concept (POC) attacks that demonstrate the need for certain additional protection mechanisms, since this helps to get wider acceptance from kernel subsystem maintainers.

[4] kernsec.org/wiki/index.php/Kernel_Self_Protection_Project

The Linux kernel Berkeley Packet Filter (BPF) Just-In-Time (JIT) compiler has been an important focus of the project, since it is widely used in the kernel and has seen successful attacks in the past. In 2012, the first JIT spray attack against the Linux BPF/JIT compiler was presented. Consequently, some countermeasures were implemented in the Linux kernel from version 3.10 that rendered this attack unsuccessful. The main measure was randomization of the memory offset where BPF programs are allocated, therefore making it difficult for an attacker to locate BPF programs in memory. However, as we show in this paper, this protection can be easily bypassed by further adjusting the attack payload and taking advantage of specific features of the randomization algorithm.

In this paper we make the following contributions:

- **Analysis and characterization** of **original POC attack** against BPF/JIT as well as proposed countermeasures (Section 3).
- **Design, implementation and analysis of our new attack** that shows how these countermeasures can be circumvented on a modern Linux 4.4 kernel (Section 4).
- **Overview of recent measures** that have been added to the Linux kernel from version 4.7 to eliminate these types of attacks altogether (Section 6).

2 Berkeley Packet Filter and JIT compiler

The need for fast network packet inspection and monitoring was obvious in early versions of UNIX with networking support. In order to gain speed and avoid unnecessary copying of packet contents between the kernel and user spaces, the notion of a kernel *packet filter* agent was introduced [20,19]. Different UNIX-based OSes implemented their own versions of these agents. The solution later adopted by Linux was the BSD Packet Filter introduced in 1993 [18], which is referred to as Berkeley Packet Filter (BPF). This agent allows a userspace program to attach a single filter program onto a socket and limit certain data flows coming through the socket in a fast and effective way.

Linux BPF originally provided a set of instructions that could be used to program a filter: this is nowadays referred to as *classic BPF* (cBPF). Later a new, more flexible, and richer set was introduced, which is referred to as *extended BPF* (eBPF) [7,21]. In order to simplify the terminology throughout this paper, we refer to the latter instruction set simply as *BPF instructions*. Linux BPF can be viewed as a minimalistic virtual machine construct [7] with a few registers, a stack and an implicit program counter. Different operations are allowed inside a BPF program, such as fetching data from the packet, arithmetic operations using constants and input data, and comparison of results against constants or packet data. The Linux BPF subsystem has a special component, called `verifier`, that is used to check the correctness of a BPF program; all BPF programs must approved by this component before they can be executed. `Verifier` is a static code analyzer that walks and analyzes all branches of a BPF program; it tries to detect unreachable instructions, out of bound jumps, loops etc. `Verifier` also enforces the maximum length of a BPF program to 4096 BPF instructions [21].

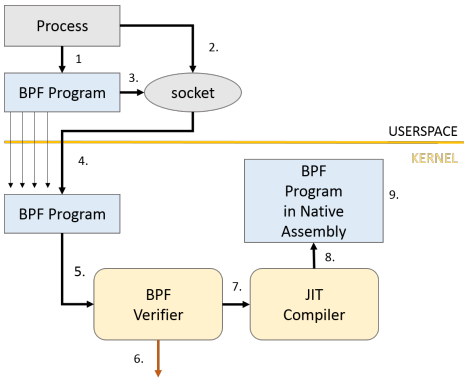


Fig. 1. Typical flow of a BPF program

While originally designed for network packet filtering, nowadays Linux BPF is used in many other areas, including system call filtering in seccomp [3], tracing [23] and Kernel Connection Multiplexer (KCM) [11].

In order to improve packet filtering performance even further, Linux utilizes a Just-In-Time (JIT) compiler [10,21] to translate BPF instructions into native machine assembly. JIT support is provided for all major architectures, including x86 and ARM. This JIT compiler is not enabled by default on standard Linux distributions, such as Ubuntu or Fedora, but it is typically enabled on network equipment such as routers.

Figure 1 shows a simplified view of how a BPF program is loaded and processed in the Linux kernel. First, a userspace process creates a BPF program, a socket, and attaches the program to the socket (steps 1-3). Next, the program is transferred to the kernel, where it is fed to `verifier` to be checked (steps 4-5). If the checks fail (step 6), the program is discarded and the userspace process is notified of the error; otherwise, if JIT is enabled, the program gets processed by the JIT compiler (step 7). The result is the BPF program in native assembly, ready for execution when the associated socket receives data (steps 8-9). The program is placed in the kernel's *module mapping space*, using the `vmalloc()` kernel memory allocation primitive.

3 JIT spray attack

JIT spraying is an attack where the behavior of a JIT compiler is (ab)used to load an attacker-provided payload into an executable memory area of the system [5]. This is usually achieved by passing the payload instructions encoded as constants to the JIT compiler and then using a suitable system vulnerability to redirect execution into the payload code. Normally the exact location of the payload is not known or controlled by the attacker, and therefore many copies of the payload are "sprayed" into OS memory to maximize the chance of success.

JIT spray attacks are dangerous because JIT compilers, due to their nature, are normally exempt from various data execution prevention techniques, such as NX bit support (known as XD bit in x86 and XN bit in ARM). Another feature that makes JIT spray attacks especially successful on the x86 architecture is its support for *unaligned instruction execution*, which is the ability to jump into the middle of a multi-byte machine instruction and start execution from there. The x86 architecture supports this feature since its instructions can be anything between 1 and 15 bytes in length, and the processor should be able to execute them all correctly in any order [2]. The first attack that introduced the notion of JIT spraying and used this technique to exploit the Adobe Flash player on Windows was done by Dion Blazakis in 2010 [6].

3.1 Original JIT spray attack on Linux

The original JIT spray attack against the Linux kernel using the BPF JIT compiler was presented by McAllister in 2012 [17]. The proof-of-concept (POC) exploit code[5] used a number of steps to obtain a root shell on a Linux device.

Creating the BPF payload: The POC creates a valid BPF program crafted to contain a small payload, comprised of the Linux kernel function calls `commit_creds(prepare_kernel_cred(0))`. This is a very common way for exploits to obtain `root` privileges on Linux: the combination of these function calls sets the credentials of the current process to superuser (username `root` and uid `0` in Linux). The payload is located at the beginning of the BPF filter program (after a fixed offset containing the BPF header and other mandatory parts of a BPF program), and must be executed starting from its first byte in order for the attack to succeed. The addresses of the `commit_creds` and `prepare_kernel_cred` kernel symbols are resolved at runtime using the `/proc/kallsyms` kernel interface exposed through the `/proc` filesystem. The payload instructions are embedded into the BPF program using a standard BPF instruction: the *load immediate* instruction (`BPF_LD+BPF_IMM`), which loads a 4 byte constant into a standard register (`eax` by default for the x86 architecture). When compiled to native assembly on x86, this instruction is transformed into a `mov $x, %eax` instruction, which corresponds to the byte sequence "`b8 XX XX XX XX`", where `b8` is the instruction opcode and the following 4 bytes are the instruction argument `$x` as an immediate value. While the attacker is able to set these 4 bytes freely, in practice only the first 3 can be arbitrarily chosen; the last byte needs to be defined so that when combined with the following `b8` instruction opcode during unaligned execution, it produces a harmless instruction. For this purpose, the last byte is chosen to be `a8`: the `a8 b8` byte sequence represents the harmless `test $0xb8, %al` x86 instruction. When the BPF load immediate instruction is repeated multiple times, this results in the byte sequence "`b8 XX XX XX a8 b8 XX XX XX a8 b8 . . .`". Figure 2 shows how the BPF program is transformed from BPF code to x86 machine code using the JIT compiler, and how the machine code looks like when an attacker succeeds in triggering unaligned

[5] github.com/kmcallister/alameda

execution from the second byte of the BPF program. The last 3 bytes of the constant (that are used for the payload) are shown in red, and it is easy to see how they get propagated from the BPF program code to the x86 machine code.

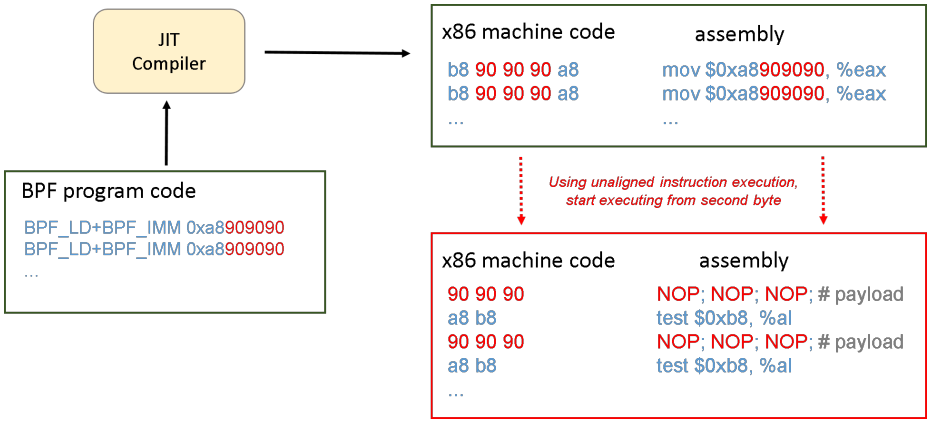


Fig. 2. BPF payload JIT compilation and unaligned execution. This sample payload contains six `NOP` instructions.

Loading the payload in memory: In order to load many copies of the BPF program payload in kernel memory, the attacker's process needs to create many local sockets, since each socket can have only one BPF program attached to it. While Linux limits the number of open file descriptors that a process can posses at any given time, McAllister used a special trick to circumvent this limit: open one local Unix socket and send the resulting file descriptor over another local Unix socket, and then close the original socket. Linux does not free the memory for the closed socket, since this might be still read by a process that receives it on the other end. Therefore, the socket is not counted towards the process socket limit, but it is kept in kernel memory regardless. By using this clever trick McAllister managed to create 8000 sockets, and correspondingly load 8000 BPF programs containing the payload in kernel memory.

Redirection of execution: Last, the proof of concept code contains a tiny kernel module (`jump.ko`) that jumps to the address specified by a userspace process using the interface provided by the `/proc` virtual filesystem. This extremely insecure module models the presence of an actual kernel bug that an attacker can use to redirect the execution flow during a real attack. `jump.ko` needs to be loaded using root privileges before the attack, which is obviously impossible for an attacker looking to obtain root privileges. This kernel module is simply used to provide an entry point to demonstrate that the JIT spray attack works.

The attack: After the attacker's program populates the kernel memory with 8000 filters containing the payload, it starts a loop where it attempts to jump to a random page within the kernel's *module mapping space* and execute the

payload at a predefined offset. The key to the attack's success is the fact that, in kernels older than 3.10, the BPF JIT compiler allocated each BPF program at the beginning of a memory page: since the length of the BPF program is fixed, the attacker always knows the correct offset to jump to on a page in order to hit the payload. Figure 3 illustrates this. One thing to note is that all memory allocations of BPF programs are done using the `vmalloc()` kernel function, which leaves a one page gap of uninitialized memory between subsequent allocations; this is done in order to protect against overruns [13].

Each guessing attempt is done by a child process: this way, in the likely case of landing on the wrong page and executing some invalid instruction, only the child process is terminated by the Linux kernel, and the parent process can continue the attack.

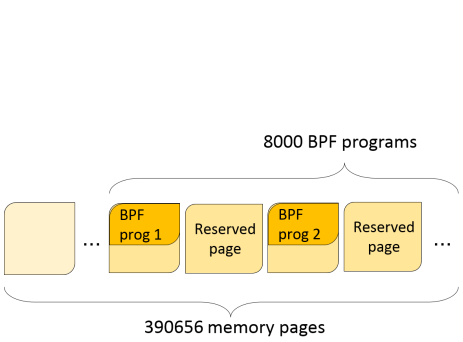


Fig. 3. Original attack

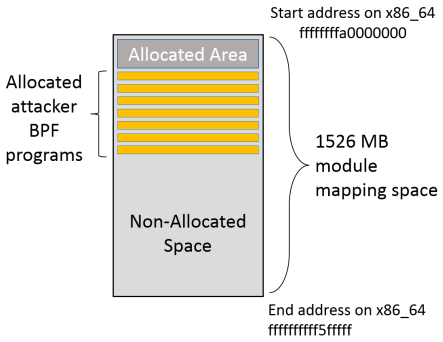


Fig. 4. BPF program allocations on `x86_64`

The kernel's *module mapping space* for `x86_64` is 1526 MB (or 390656 4KB pages)[6]. The attacker has populated 8000 4KB pages with the payload code. The resulting success probability for a single guess is

$$P_{pre} = \frac{\#\ of\ pages\ containing\ the\ payload}{\#\ of\ kernel\ module\ mapping\ pages} = \frac{8000}{390656} \approx 2\%$$

which is enough to make the multi-guess attack successful in most of the cases. It is important to note that when an attacker jumps to a page that doesn't contain the attacker's BPF program, the machine behavior is unpredictable. There are three possible cases:

- *Invalid instruction.* If the landing instruction is invalid, the child process is simply killed and the attack can continue.
- *Valid instruction with bad consequences.* If the landing instruction tampers with some key machine register, the whole OS can hang and the machine

[6] www.kernel.org/doc/Documentation/x86/x86_64/mm.txt

needs a hard reboot to recover. It is hard to estimate the theoretical probability of executing such an instruction: it depends on CPU state, content of registers and the specific instruction itself.
– *Valid instruction with no visible consequences.* It is also possible that executing a valid instruction doesn't harm the system, and the process continues to the following instruction, where all three cases are again possible.

3.2 Community response

After the attack was publicly released, the upstream Linux kernel merged a set of patches that randomized the loading address of a BPF program inside a page: instead of starting at the beginning of a page, the BPF program would be located at a random offset inside the page. In addition, the space between the page start and the BPF program - called *hole* - is filled with architecture specific instructions that aim to hang the machine if executed by an attacker. For x86, the hole is filled with repeated `INT3` (`0xcc`) instructions, which cause SIGTRAP interrupts in the Linux kernel [2]. This approach made the success probability of the attack much lower, because now the attacker needs to not only guess the correct page, but also the correct offset inside the 4KB page where the BPF program starts. This is important because in order for the attack to succeed, the attacker needs to start execution exactly at the first byte of the attack payload. The resulting success probability for a single guess dropped to

$$P_{post} = P_{pre} \cdot \frac{\#\ of\ correct\ locations\ in\ a\ page}{\#\ of\ locations\ in\ a\ page} = \frac{8000}{390656} \cdot \frac{1}{4096} \approx 0.0004\%$$

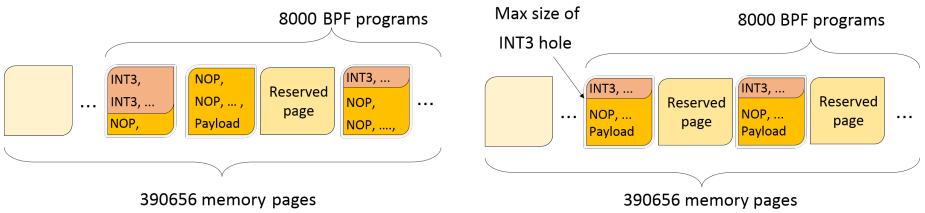
Furthermore, when the attacker jumps to a page that contains a copy of the BPF program, guessing the wrong offset is likely to be heavily punished (executing the `INT3` instruction will, in practice, result in a kernel panic and OS freeze), considerably slowing down the attacker.

4 Our attack

As part of the Kernel Self Protection Project together with one of the kernel BPF/JIT maintainers, we started to look into further securing BPF/JIT: our objective was to show that the existing measures implemented in the upstream kernel were not enough to stop JIT spray attacks.

The main part of the work was done at the end of 2015/beginning of 2016, on Ubuntu 15.10 with the latest available stable kernel at that time (`4.4.0-rc5`) compiled with default Ubuntu configuration, running in a KVM-driven virtual machine. The whole setup was done for the x86_64 architecture.

We developed two different attack approaches, discussed below. One issue common to both approaches was the inability to obtain the location of kernel symbols (specifically `commit_creds` and `prepare_kernel_cred`, needed for the attack) using the `/proc/kallsyms` kernel interface. This is because the 4.4 kernel



(a) Approach 1 (b) Approach 2

Fig. 5. Our attack

already implements kernel pointer protection, which hides the values of kernel pointers to userspace applications. This can be disabled by explicitly setting the `kptr_restrict` option to `0`; however, this operation requires root privileges. One way to overcome this difficulty is to hardcode the addresses of these symbols for a specific kernel version, after obtaining them on a machine with `kptr_restrict` disabled. This is possible on Ubuntu and similar distributions with a 4.4 kernel, since they do not utilize KASLR yet, and kernel symbols are located at a deterministic address for all copies of a specific compiled kernel (*e.g.* `4.4.0-rc5`). Then, at runtime, our attack can just resolve the correct symbol addresses by looking up the machine kernel version in a table.

4.1 Approach 1

While the size of a BPF program is limited to 4096 BPF instructions [21], this is more than enough to to obtain a compiled BPF program larger than the 4KB of a kernel memory page. When the BPF program size grows beyond one page but is under 2 pages, jumping to a page containing the attacker's BPF program has a 50% chance of success. The probability could be even higher if we extended the BPF program to be longer than 2 pages, by increasing the number of BPF instructions to the maximum value of 4096. In Approach 1, we changed the original attack to generate longer BPF programs, containing 1318 BPF instructions, which take 2 4KB pages: we did this by filling the payload with NOP instructions and placing the actual attack code at the end. Figure 5 (a) illustrates this attack approach.

When we jump to a random page, we try to execute the first 2 offsets (0 and 1) before moving to the next random page. This protects us from the unlucky case where our selected jump destination contains the `b8` byte deriving from the BPF load immediate instruction: jumping to `b8` would mean executing not the payload, but the actual `MOV %eax, XXXXXXXX` instruction. Jumping to two adjacent offsets guarantees that at least one of them will not contain `b8`. The number of sockets, and therefore loaded BPF programs, is the same (8000) as in the original attack. If we happen to jump to a page which does contain a copy of

the BPF program, but not at the beginning of the page, we hit the hole padded with `INT3` instructions, which leads to a VM hang and causes our attack to fail.

The success probability for a single guess of our Approach 1 attack can be calculated as follows. Having 8000 filters, each occupying two pages, results in 16000 pages in total. Out of these, in the worst case only half (those which contain BPF program code at the beginning) are usable for the attack: in the worst case, whenever we hit a page starting with `INT3` instructions, we hang the VM, failing the attack. Therefore, the number of usable pages drops to 8000: in other words, with Approach 1, the attacker can restore his success probability for a single guess to the original value (Section 3.1), despite the new fix.

$$P_{app1} = \frac{0.5 \cdot \#\ of\ pages\ containing\ the\ payload}{\#\ of\ kernel\ module\ mapping\ pages} = \frac{8000}{390656} \approx 2\%$$

However, the probability of a bad guess resulting in VM hang (jumping to one of the 8000 pages containing `INT3` instructions) is approximately the same 2%; this unfortunately renders the attack not useful in practice.

4.2 Approach 1 improved

To raise the success rate of Approach 1, we used the following observations.

The memory for BPF programs is allocated using the `module_alloc()` function, that in turn uses the `vmalloc()` function to allocate memory regions from the kernel's *module mapping space*. When we allocate 8000 filters, these allocations happen to be mostly adjacent in memory, and start after already allocated areas. Figure 4 illustrates this. For all x86_64 kernels, the kernel's *module mapping space* starts at the address `0xffffffffa0000000`. Given a certain kernel version and a default setup (loaded modules, filters *etc.*), the location where at least some of the attacker's filters are going to be loaded is rather predictable. For example, for our `4.4.0-rc5` kernel compiled with the default Ubuntu configuration and with no additional modules or filters loaded, there are always BPF programs allocated from the `0xffffffffa1000000` address. For the `4.4.0-36-generic#55-ubuntu` kernel provided with Ubuntu 16.04, with no additional modules or filters loaded, the allocations start around `0xffffffffc0000000`.

Knowing this, we can further narrow down the range of pages where we want to search for our payload. Instead of exploring all 390656 pages within the 1536 MB *module mapping space*, we only search through the address range from `0xffffffffa1000000` to `0xffffffffa1fff000`), which corresponds to 4095 pages - a significant reduction. Ideally, an attacker would like to have all 4095 pages allocated with filters, because this would increase the success rate. However, as already explained in Section 3.1, all memory allocations of BPF programs are done by the `vmalloc()` kernel function, which leaves a one page gap of uninitialized memory between subsequent allocations. Therefore, this smaller region only contains 1366 two-page filters, for a total of 2730 pages, and in the worst case only 1366 useful pages for an attacker. Following these observations, the attack achieves the following success probability for a single guess:

$$P_{app1_{improved}} = \frac{0.5 \cdot \#\ of\ pages\ containing\ the\ payload}{\#\ of\ pages\ in\ the\ search\ region} = \frac{1366}{4095} = 33.4\%$$

At the same time, this smaller region still contains 1366 pages likely to start with `INT3` instructions. Therefore, the probability to jump to one of these pages is the same as the success probability of a single guess. The remaining 1363 pages from the smaller region are in an uninitialized state, and jumping there equals to jumping to a randomly initialized memory location.

4.3 Approach 2

Our second approach is based on how the allocation of a BPF program happens and how the random offset of a BPF program is computed. This is done by the `bpf_jit_binary_alloc()` function, implemented in the `kernel/bpf/core.c` file.

The function first calculates the total memory size to be allocated for a program (line 223), where `proglen` is the actual BPF program length in bytes, `sizeof(*hdr)` is 4 bytes and `PAGE_SIZE` is 4096 bytes. Next, all this space is pre-filled with illegal architecture-dependent instructions (`INT3` for x86) (line 229). The actual starting offset of the BPF program is calculated last (line 234). What can be deduced from the above steps is that if we can make `proglen` to be `PAGE_SIZE - 128 - sizeof(*hdr)`, we will end up with only one page allocated for the BPF program, with a max hole size of 128 located right at the beginning of a page. While the actual size of the hole is random, the maximum size (128) is static: jumping at offset 132 (128 + `sizeof(*hdr)` will guarantee landing on the payload. This way we can fully avoid the inserted `INT3` instructions and their negative impact. Figure 5 (b) illustrates this attack approach.

In our experiments, we were able to bring the BPF program size to 3964 bytes and successfully jump over the first 132 bytes, called `hole_offset`. As in Approach 1, trying both `hole_offset` and `hole_offset + 1` protects us from the unlucky case where our selected jump destination contains the `b8` byte. The attack success probability for a single guess can be calculated as follows and equals to the original attack success rate:

$$P_{app2} = \frac{pages\ containing\ the\ payload}{kernel\ module\ mapping\ pages} = \frac{8000}{390656} \approx 2\%$$

Since - as explained above - the attacker can avoid jumping into pages containing `INT3` instructions, the global success rate of the attack is increased significantly, as shown in Section 5. The single guess success probability could be increased even further by optimizing Approach 2 with the same techniques applied to Approach 1 improved: restricting the search region to assumed BPF program locations in specific kernel versions, the single guess success probability could be increased to 50%. However, as in Approach 1 improved, this would come at the cost of a loss of generality; since the success rate for Approach 2 is already sufficiently high, we do not need to perform any such optimization which needs to be updated to each kernel version.

5 Experimental evaluation

While the theoretical single guess success probability for each attack approach is interesting on its own, what matters in practice is the global attack success rate, *i.e.* the probability for an attacker to obtain root privileges before hanging the VM. This success rate is hard to theoretically calculate, due to the difficulty of characterizing the behavior of several factors which influence it; these factors are - for example - the CPU state and kernel memory layout at the time of the attack, the specific random number generator used (which may generate random numbers according to a non-ideal distribution), *etc.* Therefore, we experimentally evaluated our different attack approaches using the following setup.

The attack was run in a Linux virtual machine powered by `qemu`, running the `4.4.0-36-generic#55-ubuntu` kernel provided with Ubuntu 16.04. The virtual machine had 2048MB of RAM and access to host 2 CPUs (Intel Core i7-3520M). We collected many attack runs per each approach: each run terminates either with an attack success (the attacker's process obtains root privileges) or by an attack failure (the VM hangs). If an attacker's process succeeds in obtaining root privileges, it is terminated, all filters are unloaded and a new run is performed from scratch. If the VM hangs, the virtual machine is forcefully restarted and a new run is performed. The measurements are shown in Table 1.

Attack	***# guesses***	***P single guess***	***# runs***	***P global***	***Mean(ℓ)***	***SD(ℓ)***	***Median(ℓ)***
4.1	12280	2.0 %	410	58.3 %	29.6	28.2	21
4.2	858	33.2 %	438	65.1 %	2.1	1.5	2
4.3	80190	2.0 %	1636	99.6 %	49	47.3	35

Table 1. Experimental results. ℓ is the length of a successful run.

The single guess success rate is calculated as the number of successful guesses divided by the total number of guesses. The global attack success success rate is calculated as the number of successful runs divided by the total number of runs. The mean and standard deviation of run lengths are calculated over the lengths of successful runs.

6 Mitigation measures

Our attack demonstrated conclusively that randomization alone is insufficient to stop BPF JIT spray attacks. The main reason is that randomization does not address the underlying problem of the attacker being able to deterministically control what instruction would be executed when the attacker can jump to a location in the payload of the BPF program and trigger unaligned execution.

Right after the original JIT spray attack in 2012, the Grsecurity[7] kernel security project released a blinding-based hardening mechanism to defend against

[7] grsecurity.net

the attack. Their implementation only supported the `x86` architecture. At that time it was not merged into the upstream Linux kernel due to the desire to have an architecture independent approach, the performance implications of the feature, and most of all the belief that the randomization measures implemented in the upstream kernel would be enough to stop BPF JIT spray attacks.

Since our attack was demonstrated, a number of mitigation measures have been merged to the upstream kernel. The main measure is blinding the constants in eBPF[8]. Blinding consists of XORing each constant with a random number at compile time, and XORing it with the same random number immediately before using it at runtime. This way, constants never appear in memory "in clear", and an attacker cannot deterministically control the content of the memory area in which they are stored. Blinding functionally turns the attacker's payload code into a set of random values, and therefore blocks the code injection which allowed the JIT spray attack: the probability of finding instructions that can lead to the behavior desired by the attacker is equal to the probability of finding them in a randomly initialized memory area. The feature changes the BPF program code in the following way, shown in Figure 6. First, a new random number is generated for each constant; this number is XORed with the constant, and the result is used as the parameter of a `BPF_MOV` instruction added to the BPF program. This means that, at runtime, the XORed constant will be written to a register. Then, an instruction is added to the BPF program to XOR the same random number with this register: this is used to recover the original value of the constant at runtime. Once recovered, the original value of the constant is then used by the actual instruction which contained the constant in the original code. This is achieved by modifying the original instruction to operate on the freshly recovered value from the register, and not on an immediate constant. As a result, the x86 assembly code produced by the JIT compiler does not contain any constants in clear anymore.

In contrast to the `x86`-specific Grsecurity implementation, the new implementation is architecture independent: this is obtained by blinding constants already at the eBPF instruction level, and feeding the blinded constants to the JIT compiler. This implementation does require minimal support from each architecture that would like to enable this security feature, but the required support is straightforward. This not only allows a unified and solid design for BPF/JIT hardening for all architectures, but also further improves security by having a single, well-reviewed hardening implementation. The performance overhead of blinding was measured to be around 8% for the test suite cases, and expected to be even smaller in real world scenarios[9]. This protection has been merged to the upstream kernel in May 2016 and was released as part of version 4.7.

More hardening has been done to prevent exploiting Unix domain sockets. A patch[10] has been merged to prevent circumventing the resource limit on the amount of open file descriptors, which was released in kernel version 4.5.

[8] git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=4f3446b
[9] patchwork.ozlabs.org/patch/622075/
[10] git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=712f4aa

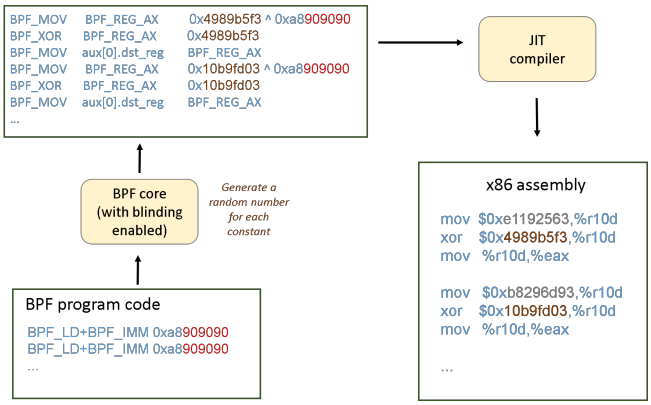


Fig. 6. BPF constant blinding. The attacker payload is shown in red; the random numbers used for blinding are in brown; the blinded constants are in grey.

Kernel Address Space Layout Randomization (KASLR) for x86_64, an important feature that aims to prevent exploits from relying on static locations of kernel symbols, has been released in the kernel as part of version 4.8. If enabled, this feature randomizes the physical and virtual memory location of where the kernel image is decompressed, and makes it significantly harder for attackers to discover the location of kernel symbols needed for attacks. For example, it is not possible anymore to rely on binary-specific locations of `commit_creds()` or `prepare_kernel_cred()` symbols based on kernel version. An attacker would have to instead use various information leaks to obtain these values [15]. While KASLR is important, it still does not provide full protection from all exploits. For example, the addresses where `vmalloc()` allocates kernel memory are still not randomized, which can provide additional information to improve an attack.

7 Related work

With the development of various hardening mechanisms, such as stack protectors [12], DEP [1] and ASLR [24], it became harder for attackers to perform code injection and code reuse attacks. However, if a system features a JIT compiler, this provides an attractive attack vector: the JIT compiler produced code can be largely controlled or predicted by an attacker, and it is executable by default. Since the first successful JIT spray attack on Adobe Flash Player on Windows [6], many similar attacks have been done on different JIT compilers. At the same time, a number of JIT hardening mechanisms have been proposed. Chen *et al.* [8] proposed *JITDefender*, a system for userspace JIT engines that marks all JIT-produced code as non-executable and only marks it executable during the actual execution by the JIT engine. This design is not applicable to the kernel BPF JIT engine, since there may be many different small BPF programs loaded at the same time in memory, and their execution would need to

be constantly enabled/disabled depending on each passing packet and allocated sockets. Homescu *et al.* [14] developed the userspace *librando* library, which, in addition to performing constant blinding, post-processes JIT-emitted code in order to randomize and diversify its location. Athanasakis *et al.* [4] demonstrated that JIT spray is still possible if constants of 2 bytes or less are left unblinded. They performed their attack successfully on the JIT engines of Mozilla Firefox and Microsoft Internet Explorer. Their conclusion was that blinding all constants regardless of their size stops the attack, but this measure was found to be too performance costly for the mentioned userspace JIT engines. Jangda *et al.* [16] propose the *libsmack* library as an alternative to constant blinding. Their idea is to replace each constant with a randomized address that in turn stores the value of the constant. The library demonstrates slightly better performance compared to simple blinding of constants. In some cases, in order to optimize the performance, the code produced by a JIT engine is made both executable and temporarily writable in a cache. Song *et al.* [22] showed that this can be successfully exploited by an attacker through code cache injection techniques.

Fortunately, many of the above problems and challenges are not applicable to the Linux Kernel BPF JIT compiler. The JIT-produced code is only executable - and never writable - after it has been placed in memory. Since the size of each BPF constant is fixed to 4 bytes regardless of its actual value, it is only possible to apply full blinding; there is no possibility of obtaining a speed-security tradeoff by applying partial blinding. This allows the hardening solution of blinding all constants to be simple and fully effective, relinquishing the need for any additional measures. One additional reason why the mechanisms above do not directly apply to the BPF JIT case is given by the position of the BPF engine. Since the BPF JIT engine is implemented inside the Linux kernel, any security solution must be integrated into the engine itself instead of using external libraries or post-processing mechanisms.

8 Conclusions

In this paper we presented two different approaches to successfully attack the BPF/JIT engine in the Linux kernel up to version 4.4. We demonstrated that randomization alone is insufficient to stop BPF JIT spray attacks, since it does not remove the attacker's ability to supply the attack payload using BPF constants. As a result of our attack, a robust constant blinding solution against BPF JIT spray attacks has been merged to the upstream Linux kernel, which fixes the actual root cause of the problem. More information about the attack can be obtained from our project page[11].

Acknowledgments. The authors would like to thank Daniel Borkmann for his helpful discussions about BPF/JIT, and his readiness and enthusiasms to make it more secure.

[11] ssg.aalto.fi/projects/kernel-hardening

References

1. A detailed description of the Data Execution Prevention (DEP) feature. support.microsoft.com/en-us/kb/875352 (2016)
2. Intel® 64 and IA-32 Architectures Software Developer's Manual. www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf (2016)
3. SECure COMPuting with filters. www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt (2016)
4. Athanasakis, M., et al.: The Devil is in the Constants: Bypassing Defenses in Browser JIT Engines. In: NDSS (2015)
5. Bania, P.: JIT spraying and mitigations. arXiv preprint arXiv:1009.1038 (2010)
6. Blazakis, D.: Interpreter Exploitation: Pointer Inference and JIT Spraying. www.semantiscope.com/research/BHDC2010/BHDC-2010-Paper.pdf (2010)
7. Borkmann, D.: On getting tc classifier fully programmable with cls_bpf. www.netdevconf.org/1.1/proceedings/papers/On-getting-tc-classifier-fully-programmable-with-cls-bpf.pdf (2016)
8. Chen, P., Fang, Y., Mao, B., Xie, L.: JITDefender: A defense against JIT spraying attacks. In: IFIP. pp. 142–153 (2011)
9. Cook, C.: Status of the Kernel Self Protection Project. outflux.net/slides/2016/lss/kspp.pdf (2016)
10. Corbet, J.: A JIT for packet filters. lwn.net/Articles/437981/ (2012)
11. Corbet, J.: The kernel connection multiplexer. lwn.net/Articles/657999/ (2015)
12. Edge, J.: "Strong" stack protection for GCC. lwn.net/Articles/584225/ (2014)
13. Gorman, M.: Understanding the Linux virtual memory manager (2004)
14. Homescu, A., Brunthaler, S., Larsen, P., Franz, M.: Librando: transparent code randomization for just-in-time compilers. In: CCS. pp. 993–1004 (2013)
15. Jang, Y., Lee, S., Ki, T.: Breaking Kernel Address Space Layout Randomization with Intel TSX. In: CCS. pp. 380–392 (2016)
16. Jangda, A., Mishra, M., Baudry, B.: libmask: Protecting Browser JIT Engines from the Devil in the Constants. In: PST (2016)
17. McAllister, K.: Attacking hardened Linux systems with kernel JIT spraying. mainisusuallyafunction.blogspot.de/2012/11/attacking-hardened-linux-systems-with.html (2012)
18. McCanne, S., Jacobson, V.: The BSD Packet Filter: A New Architecture for User-level Packet Capture. In: USENIX winter. vol. 46 (1993)
19. Mogul, J.: Efficient use of workstations for passive monitoring of local area networks, vol. 20. ACM (1990)
20. Mogul, J., Rashid, R., Accetta, M.: The packer filter: an efficient mechanism for user-level network code, vol. 21. ACM (1987)
21. Schulist, J., et al.: Linux Socket Filtering aka Berkeley Packet Filter (BPF). www.kernel.org/doc/Documentation/networking/filter.txt (2016)
22. Song, C., Zhang, C., Wang, T., Lee, W., Melski, D.: Exploiting and Protecting Dynamic Code Generation. In: NDSS (2015)
23. Starovoitov, A.: Tracing: attach eBPF programs to kprobes. lwn.net/Articles/636976/ (2015)
24. Team, P.: PaX address space layout randomization (ASLR) (2003)

Publication VI

Reshetova, Elena and Liljestrand, Hans and Paverd, Andrew and Asokan, N. Towards Linux Kernel Memory Safety. Submitted to *ACM Conference on Data and Application Security and Privacy*, March 2018.

Towards Linux Kernel Memory Safety

Submission 44

ABSTRACT

The security of billions of devices worldwide depends on the security and robustness of the mainline Linux kernel. However, the increasing number of kernel-specific vulnerabilities, especially memory safety vulnerabilities, shows that the kernel is a popular and practically exploitable target. Two major causes of memory safety vulnerabilities are reference counter overflows (temporal memory errors) and lack of pointer bounds checking (spatial memory errors).

To succeed in practice, security mechanisms for critical systems like the Linux kernel must also consider performance and deployability as critical design objectives. We present and systematically analyze two such mechanisms for improving memory safety in the Linux kernel: (a) an overflow-resistant reference counter data structure designed to accommodate typical reference counter usage in kernel source code, and (b) runtime pointer bounds checking using Intel MPX in the kernel.

CCS CONCEPTS

• **Security and privacy → Operating systems security**;

KEYWORDS

Linux kernel, memory safety

ACM Reference format:
Submission 44. 2018. Towards Linux Kernel Memory Safety. In *Proceedings of ACM Conference on Data and Application Security and Privacy, Tempe, Arizona USA, March 2018 (CODASPY'18),* 12 pages.
https://doi.org/10.475/123$_4$

1 INTRODUCTION

The Linux kernel lies at the foundation of millions of different devices around us, ranging from servers and desktops to smartphones and embedded devices. While there are many solutions for strengthening Linux application security covering access control frameworks (SELinux [42], AppArmor [4]), integrity protection systems (IMA/EVM [17], dm-verity [13]), encryption, key management and auditing, they are rendered ineffective if an attacker can gain control of the kernel. Recent trends in Common Vulnerabilities and Exposures (CVEs) indicate a renewed interest in exploiting the kernel [27]. Kernel bugs are long-lived, on average taking five years before being found and fixed [9], and even when fixed, security updates might not be deployed to all vulnerable devices. Thus, we cannot rely solely on retroactive bug fixes, but need proactive measures to harden the kernel by limiting its exploitability. This is the goal of the Kernel Self Protection Project (KSPP) [21], a large community of volunteers working on the mainline Linux kernel. In this paper, we describe our contributions, as part of the KSPP, to the development of two recent kernel memory safety mechanisms.

Depending on their severity, memory errors can allow an attacker to read, write, or execute memory, thus making them attractive targets. For example, use-after-free errors and buffer overflows feature prominently in recent Linux kernel CVEs [6, 37]. Memory errors arise due to the lack of inherent memory safety in C, the main implementation language of the Linux kernel. There are two fundamental classes of memory errors:

Temporal memory errors occur when pointers to uninitialized or freed memory are dereferenced. One common case is a *use-after-free* error e.g., dereferencing a pointer that has been prematurely freed by another execution thread. A null pointer dereference error, although common, is more challenging to exploit [48], whereas use-after-free errors are both exploitable and common, and have featured in CVE-2014-2851, CVE-2016-4558, and CVE-2016-0728, among others.

Spatial memory errors occur when pointers are used to access memory outside the bounds of their intended areas. A prime example is a buffer overflow, which occurs when the amount of data written exceeds the size of the buffer. Spatial memory errors have appeared in CVE-2014-0196, CVE-2016-8440, CVE-2016-8459, and CVE-2017-7895.

While memory safety has been scrutinized for decades (Section 7), much of the work has focused on user-space. These solutions are not readily transferable to kernel-space. For instance, solutions to protect against spatial memory errors use a static and fast addressing scheme to store pointer bounds by treating virtual memory as an endless physical memory [26, 38]. This approach is not viable in kernel-space as it cannot trivially handle arbitrary page faults. Although there have been a few proposals for improving kernel memory safety (e.g., kCFI [40] and KENALI [44]), these have not considered the critical issue of deployability in the mainline Linux kernel. Other mechanisms, such as the widely used Kernel Address Sanitizer (KASAN) [19], are intended as debugging facilities, not runtime protection.

In this paper, we present solutions for mitigating the major causes of both temporal and spatial memory errors: (1) we contributed to the design of `refcount_t` a new reference counter data type that prevents reference counter overflows; in particular, we analyzed how developers have used reference counters in the kernel, and based on this we designed extensions to the new `refcount_t` API; and (2) we developed a mechanism for performing efficient pointer bounds checking at runtime, using Intel Memory Protection Extensions (MPX). The new `refcount_t` data structure and API are already integrated into the Linux mainline kernel. More than half our patches (123/233 at the time of writing) to convert existing reference counters to use `refcount_t` have also been already integrated. In summary, we claim the following contributions:

CODASPY'18, March 2018, Tempe, Arizona USA

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123$_4$

- **Extended `refcount_t` API**: We present a heuristic technique to identify instances of reference counters in the kernel source code (Section 4.1). By analyzing reference counter usage in the kernel, we designed extensions to the API of the new `refcount_t` reference counter data structure (Section 4.3). We also converted the entire kernel source tree to use the new API (Section 4.6).
- **MPXK**: We present a spatial memory error prevention mechanism for the Linux kernel based on the recently released Intel Memory Protection Extensions (MPX) (Section 5).
- **Evaluation**: We present a systematic analysis of `refcount_t` and MPXK in terms of performance, security, and usability for kernel developers. (Section 6).

Our primary objective in this work was to deploy our results into the mainline Linux kernel. In Section 8 we reflect on our experience in this regard, and offer suggestions for other researchers with the same objective.

2 BACKGROUND

2.1 Linux kernel reference counters

Temporal memory errors are especially challenging in low-level systems that cannot rely on automated facilities such as garbage collection for object destruction and freeing of allocated memory. In simple non-concurrent C programs, objects typically have well defined and predictable allocation and release patterns, which make it trivial to free them manually. However, complex systems, such as the Linux kernel, rely heavily on object reuse and sharing patterns to minimize CPU and memory use.

In lieu of high-level facilities like garbage collection, object lifetimes can be managed by using ***reference counters*** to keep track of current uses of an object [8]. Whenever a new reference to an object is taken, the object's reference counter is incremented, and whenever the object is released the counter is decremented. When the counter reaches zero, it means that the object is no longer used anywhere, so the object can be safely destroyed and its associated memory can be freed. However, reference counting schemes are historically error prone; A missed increment or decrement, often in a rarely exercised code path, could imbalance the counter and cause either a memory leak or a use-after-free error.

Reference counters in the Linux kernel are typically implemented using the `atomic_t` type [25], which in turn is implemented as an `int` with a general purpose atomic API consisting of over 100 functions. This approach poses two problems. First, the general purpose API provides ample room for subtly incorrect implementations of reference counting schemes motivated by performance or implementation shortcuts. Second, as a general purpose integer `atomic_t` allows its instances to overflow. When reference counters are based on `atomic_t`, this implies that they can inadvertently reach zero via only increments. Overflow bugs are particularly hard to detect using static code analysis or fuzzing techniques because they require many consequent iterations before the overflow happens [32]. A recent surge of exploitable errors, such as CVE-2014-2851, CVE-2016-4558, CVE-2016-0728, CVE-2017-7487 and CVE-2017-8925, specifically target reference counters.

2.2 Intel Memory Protection Extensions (MPX)

Intel Memory Protection Extensions (MPX) [38] is a recent technology to prevent spatial memory errors. It is supported on both Pentium core and Atom core micro-architectures from Skylake and Goldmount onwards, thus targeting a significant range of end devices from desktops to mobile and embedded processors. In order to use the MPX hardware, both the compiler and operating system must support MPX. On Linux, MPX is supported by the GCC and ICC compilers, and the kernel supports MPX for user-space applications. MPX is source and binary compatible, meaning that no source code changes are required and that MPX-instrumented binaries can be linked with non-MPX binaries (e.g., system libraries). MPX-instrumented binaries can still be run on all systems, including legacy systems without MPX hardware. If MPX hardware is available, the instrumented binaries will automatically detect and configure this hardware during process initialization.

MPX prevents spatial memory errors by checking pointer bounds before a pointer is dereferenced. Conceptually, every pointer is associated with an upper and lower bound. Pointer bounds are determined by means of compile-time code instrumentation. For a given pointer, the bounds can either be set statically (e.g., based on static data structure sizes), or dynamically (e.g., using instrumented memory allocators). For example, `malloc` is instrumented to set the bounds of newly allocated pointers.

In order to perform a bounds check, the pointer's bounds must be loaded in one of the four new MPX bound (`bndx`) registers. When not in these registers, the MPX instrumentation stores bounds on the stack or in static memory. However, in some cases the number of bounds to be stored cannot be determined at compile time (e.g., a dynamically-sized array of pointers). In these cases, bounds are stored in memory in a new two-level metadata structure, and accessed using the new bound load (`bndldx`) and store (`bndstx`) instructions. These instructions use the address of the pointer to look up an entry in the process's Bound Directory (BD), which in turn points into a operating system managed Bound Table (BT), which stores the pointer's address and bounds. On 64-bit systems the BD is 2 GB, and each individual BT is 4 MB. To reduce this high memory overhead, the Linux kernel only allocates physical memory to the BD regions when they are written to, and only allocates individual BTs when they are accessed. When a bounds check fails, the CPU issues an exception that must be handled by the operating system.

3 PROBLEM STATEMENT

The goal of this work is to limit the Linux kernel's vulnerability to memory safety errors, with a focus on common errors and attacks. Specifically, we focus on preventing two causes of memory errors:

- **Reference counter overflows**: the security requirement is to design a reference counter type and associated API such that the counter is guaranteed never to overflow.
- **Out-of-bounds memory accesses**: the security requirement is to design an access control scheme that prevents out-of-bounds accesses.

In addition to the security requirements, the following are mandatory design considerations if the solution is to be used in practice:

- **Performance.** Any solutions added to the kernel must minimize performance impact. Some subsystems, such as networking and filesystem, are particularly sensitive to performance. Security mechanisms perceived as imposing a high performance overhead risk being rejected by the maintainers of these subsystems. Furthermore, the Linux kernel runs on a vast range of devices, including closed fixed-function devices like routers, where software attacks are not a threat but performance requirements are stringent. Thus, the solutions must adapt or be configurable depending on the needs of different usage scenarios.
- **Deployability.** Given our goal of integrating our solutions into the mainline Linux kernel, their implementation must follow the Linux kernel design guidelines, impose minimal kernel-wide changes, and be both maintainable and usable. *Usability for kernel developers* is particularly crucial for new features that may be adopted at the discretion of subsystem developers.

4 REFERENCE COUNTER OVERFLOWS

We developed a methodology to find reference counters in the kernel source code as well as analyze how they are typically used. A new type `refcount_t` and corresponding minimal API were introduced by Peter Zijlstra, one of the Linux maintainers of related subsystems. It provides various reference counter specific protections by preventing some outright incorrect behavior, i.e., incrementing on zero and overflowing the counter. Based on our analysis of reference counter use, we proposed several additions to the initial `refcount_t` API, making it widely usable as a replacement for existing reference counters and future implementations. To comprehensively prevent reference counter overflows, we have developed a set of patches to convert all conventional reference counters in the kernel to use `refcount_t`.

4.1 Analyzing Linux Kernel reference counters

Reference counters are spread across the Linux kernel. We thus systematically analyzed the source code to locate instances of reference counters based on the `atomic_t` type using Coccinelle [7], a static analyzer integrated into the Linux Kernel build system (KBuild). Coccinelle takes code patterns as input and finds (or replaces) their occurrences in the given code base. We defined three such code patterns to identify reference counters based on their behavior (see Listing 3 in the Appendix for the full patterns):

(1) Freeing a referenced object based on the return value from `atomic_dec_and_test` or one its variants. This is the archetypical reference counter use case.
(2) Using `atomic_add_return` to decrement a variable and compare its updated value, typically against zero. These use cases are essentially variations of the basic `dec_and_test` cases albeit using a different function for the implementation.
(3) Using `atomic_add_unless` to decrement a counter only when its value differs from one. This case is less common.

These patterns are strong indicators that the identified object employs an `atomic_t` reference counting scheme. So far, this approach detected all occurrences of reference counters. Some false positives were reported, particularly on implementations with `atomic_t` variables that occasionally exhibit reference counter-like behavior.

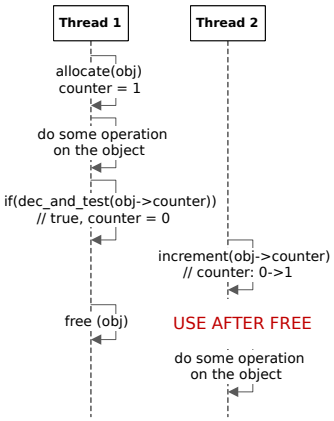


Figure 1: Potential use-after-free when incrementing a reference counter from zero.

For example, under one condition an object might be freed when such counter reaches zero (behaves like a reference counter), but under a different condition the object might instead be recycled when the counter reaches zero (see Section 4.5 for examples). Of the 250 `atomic_t` variables reported on an unmodified v4.10 kernel we have manually confirmed 233 variables as reference counters.

4.2 refcount_t API

The initial `refcount_t` API, introduced by Peter Zijlstra[1] (Listing 1), was designed around strict semantically correct reference counter use. This meant that beyond `set` and `read` calls, the API provided only two incrementing functions, both of which refuse to increment the counter when its value is zero, and three variations of decrementing functions that enforce the checking of return values via compile time warnings. When decrementing the caller needs to know if the value reached zero, which is indicated by a return value of `true`. A `true` return value from `refcount_inc_not_zero` means that the counter was non-zero, indicating that the referenced object is safe to use.

Listing 1: Initial bare `refcount_t` API.

```
void refcount_set(refcount_t *r, unsigned int n);
unsigned int refcount_read(const refcount_t *r);
bool refcount_inc_not_zero(refcount_t *r);
void refcount_inc(refcount_t *r);
bool refcount_dec_and_test(refcount_t *r);
bool refcount_dec_and_mutex_lock(refcount_t *r,
                                 struct mutex *lock);
bool refcount_dec_and_lock(refcount_t *r,
                           spinlock_t *lock);
```

The checks are performed to ensure that a reference counter reaching zero triggers object release, thus preventing a potential memory leak, whereas increment on zero is prohibited to avoid potential use-after-free, as illustrated in Figure 1.

The main challenge in designing the `refcount_t` API was how to deal with an event that would otherwise cause the counter to overflow. One approach would be to simply ignore the event, such that the counter remains at its maximum value. However, this

[1] Anonymized

means that the number of references to the object will be greater than the maximum counter value. If the counter is subsequently decremented, it will reach zero and the object will be freed before all references have been released (i.e., leading to a use-after-free error). To overcome this challenge, the `refcount_t` API instead *saturates* the counter, such that it remains at its maximum value, even if there are subsequent increment or decrement events. A saturated counter would, therefore, result in a memory leak since the object will never be freed. However, a cleanly logged memory leak is a small price to pay for avoiding the potential security vulnerabilities of a reference counter overflow. This approach is similar to that previously used by the PaX/Grsecurity patches [5].

4.3 Our extensions to the `refcount_t` API

Our efforts in finding existing reference counters revealed several variations of the strict archetypical reference counting schemes, which are incompatible with the initial `refcount_t` API. As a result, we designed several API additions that got integrated into the Linux kernel[2] . The new API calls are shown in Listing 2.

Listing 2: Our additions to the `refcount_t` API.

```
void refcount_add(unsigned int i, refcount_t *r);
bool refcount_add_not_zero(unsigned int i, refcount_t *r);
void refcount_dec(refcount_t *r);
bool refcount_dec_if_one(refcount_t *r);
bool refcount_dec_not_one(refcount_t *r);
void refcount_sub(refcount_t *r);
bool refcount_sub_and_test(unsigned int i, refcount_t *r);
```

The functions allowing arbitrary additions and subtractions, i.e., the `refcount_add` and `refcount_sub` variants, are needed in situations where larger value changes are needed. For example, the `sk_wmem_alloc` variable in the networking subsystem serves as a reference counter but also tracks transfer queue, which need arbitrary additions and subtractions. `refcount_sub_and_test` and `refcount_add_not_zero` also provide a return value that, that indicate that the counter reached the value of zero. `refcount_dec` accommodates situations where the forced return value checks would incur needless overhead. For instance, some functions in the `btrfs` filesystem handle nodes that are guaranteed to be cached, i.e., at least the reference held by the cache will be remaining. Finally, `refcount_dec_if_one` and `refcount_dec_not_one` enable schemes that require specific operations before or instead of releasing objects. For instance, the networking subsystem extensively utilizes patterns where the value of one indicates that an objects is invalid, but can be recycled.

4.4 Implementation considerations

To avoid costly lock or mutex use the `refcount_t` API generic, i.e., not architecture specific, implementation uses the *compare-and-swap* pattern, which is built around the atomic `atomic_cmpxchg` function, shown in Algorithm 1. On x86 `atomic_cmpxchg` is implemented as a a single atomic CPU instruction, but the implementation is guaranteed to be atomic regardless of architecture. The function always returns the prior value but exchanges it only if it was equal to the given condition value *comp*. The compare-and-swap works by indefinitely looping until a `cmpxchg` succeeds. This avoids costly locks and allows all but the `cmpxchg` to be non-atomic. Note that in the typical case, without concurrent modifications, the loop runs only once, thus being much more efficient than locks and potential blocking of execution.

Algorithm 1 `cmpxchg`(*atomic*, *comp*, *new*) sets the value of *atomic* to *new* if, and only if, the prior value of *atomic* was equal to *comp*. Whether the value was changed or not the function always returns the prior value of *atomic*. In practice the function is implemented as a single inline CPU instruction.

```
1: old ← atomic.value
2: if old = comp then
3:     atomic.value ← new
4: end if
5: return old
```

As an example of the extended `refcount_t` API implementation, consider the `refcount_add_not_zero` function shown in Algorithm 2. It is used to increase `refcount_t` when acquiring a reference to the associated object, a return value of `true` indicates that the counter was non-zero, and thus the associated object is safe to use. The actual value from a user perspective is thus irrelevant and `refcount_add_not_zero` guarantees only that the return value is true if, and only if, the value of `refcount_t` at the time of the call was non-zero. Internally the function further guarantees that the increment takes place only when the prior value was in the open interval $(0, UINT\_MAX)$, thus preventing use-after-free due to increment from zero and use-after-free due to overflow. This case results in the default return statement at line 19. Attempted increment from zero or `UINT_MAX` results in returns at lines 3 and 6, respectively. Finally, an addition that would overflow the counter instead saturates it by setting its value to `UINT_MAX` on line 10.

4.5 Challenges

Despite the additions to the `refcount_t` API, several reference counting instances nonetheless required careful analysis and in some cases challenging modifications to the underlying logic. These challenges were the main reason for not using Coccinelle to automatically convert reference counting schemes. The most common challenges are *object pool patterns* and unconventional reference counters.

While the object pool pattern is accommodated by our additions to the `refcount_t` API - namely, by providing functions that distinguish the value one - such implementations typically employ negative values or other means to distinguish between freed and recyclable objects. These patterns therefore often necessitated non-trivial changes to ensure that neither increment on zero operations nor negative values are expected. Overall we encountered 6 particularly challenging recycling schemes. For example, the `inode` `refcount_t` conversion spanned a total of 10 patches[3] and required so many changes that it has not yet been accepted by the maintainer.

In some cases reference counters were used in *non-conventional ways*, such as to govern other behavior or track other statistics in addition to the strict reference count itself (e.g., the network socket

[2] Anonymized

[3] Anonymized

Algorithm 2 `refcount_add_not_zero` (*refcount*, *summand*) attempts to add a *summand* to the value of *refcount*, and returns true if, and only if, the prior value was non-zero. Note that the function is not locked and only the `cmpxchg` (line 13) is atomic, i.e., the value of *refcount.value* can change at any point of execution. The loop ensures that the `cmpxchg` eventually succeeds. Overflow protection is provided by checking if the value is already saturated (line 6) and by saturating, instead of overflowing, on line 10.

Ensure: $retval = \textbf{true} \Leftrightarrow refcount.value > 0$
Ensure: $value\_unchanged \Leftrightarrow refcount.value \in \{0, UINT\_MAX\}$

```
1:  val ← refcount.value {use local copy}
2:  while true do
3:    if val = 0 then
4:      return false {counter not incremented from zero}
5:    end if
6:    if val = UINT_MAX then
7:      return true {counter is saturated, thus not zero}
8:    end if
9:    new ← val + summand {calculate new value}
10:   if new < val then
11:     new ← UINT_MAX {Saturate instead of overflow}
12:   end if
13:   old ← atomic_cmpxchg(refcount.atomic, val, new)
14:   if old = val then {if refcount.value was unchanged, then}
15:     break {value was updated by cmpxchg}
16:   end if
17:   val ← old {update val for next iteration}
18: end while
19: return true {value incremented}
```

`sk_wmem_alloc` variable mentioned above). The `refcount_t` API can be surprising or outright erroneous for such unconventional uses; it might for instance be expected that such variables can be incremented from zero or potentially reach negative values. Our conversion efforts touched upon 21 reference counters in this category.

4.6 Deployment of refcount_t

We developed 233 distinct kernel patches, each converting one distinct variable, spanning all the kernel subsystems. During our work, the `refcount_t` API, with our additions, was also finalized in the mainline Linux kernel. The next stage of our work consisted of submitting all the patches to the respective kernel maintainers and adapting them based on their feedback. Based discussions with maintainers, some patches were permanently dropped, either because they would require extensive changes in affected subsystems or would incur unacceptable performance penalty without any realistic risk of actually overflowing the particular counter. As explained in Section 3, some performance-sensitive systems proved challenging to convert due to performance concerns. As a result, a new `CONFIG_REFCOUNT_FULL` kernel configuration option was added to allow switching the `refcount_t` protections off and thus use the new API without any performance overhead. This can be utilized by devices that have high performance requirements but are less concerned about security based on their nature (e.g., closed devices such as routers that do not allow installation of untrusted software). These patching efforts, discussions, and patch reviews also uncovered related reference counting bugs that were fixed in subsequent kernel patches[4].

5 OUT-OF-BOUNDS MEMORY ACCESS

To prevent spatial memory errors in the Linux kernel, we have adapted Intel MPX for in-kernel use. MPX was designed for both user-space and kernel-space (e.g., the hardware includes separate configuration registers for each). However, until now, the Linux kernel and the GCC compiler only supported MPX for user-space applications. Our solution, *MPX for Kernel (MPXK)*, is realized as a new GCC instrumentation plugin (based on the existing MPX support in GCC). In the following subsections, we describe the various challenges of using MPX in the Linux kernel, and our respective solutions in MPXK.

5.1 Memory use

The first challenge arises from the high memory overhead incurred by the MPX Bound Directory (BD) and Bound Tables (BT). As explained in Section 2.2, user-space MPX attempts to reduce this overhead by allocating this memory only when needed. However, this requires the kernel to step in at arbitrary points during execution to handle page faults or Bound Faults caused by caused by BD dereferences or unallocated BTs. This approach cannot be used in MPXK because the kernel cannot handle page faults at arbitrary points within its own execution. It is also not feasible to pre-allocate the BD and BTs, as this would increase base memory usage by over 500% and require extensive modifications to accommodate certain classes of pointers (e.g., pointers originating from user-space). An alternative approach would be to substitute the hardware-backed BD and BTs with our own metadata, similar to SoftBound [26] or KASAN [19]. However, this would still incur the same memory and performance overheads as those systems.

To overcome this challenge, MPXK dynamically determines pointer bounds using *existing kernel metadata*. Specifically, we reuse the kernel memory management metadata created by `kmalloc`-based allocators. We define a new function, `mpxk_load_bounds`, that uses this existing metadata to determine the bounds for a pointer allocated by `kmalloc`, and loads these into the MPX registers. A side-effect of this approach is that bounds are rounded up to the nearest allocator cache size (i.e., may be slightly larger than the requested allocation size). However, this has no security implications because the allocator will not allocate any other objects in the round-up memory area [1]. MPXK thus never uses the `bndldx` or `bndstx` instructions. Our kernel instrumentation only uses `mpxk_load_bounds` when the bounds cannot be determined at compile-time (e.g., for dynamically allocated objects).

5.2 Kernel support code

In user-space, MPX is supported by GCC library implementations for various support functionality, such as initialization and function wrappers. The user-space instrumentation initializes MPX during

[4] http://lkml.org/lkml/2017/6/27/567, http://lkml.org/lkml/2017/3/28/383 etc.

process startup by allocating the BD in virtual memory and initializing the MPX hardware. However, this existing initialization code cannot be used directly in the Linux kernel because kernel-space MPX must be configured during the kernel boot process.

In user-space, the compiler also provides instrumented wrapper functions for all memory manipulation functions, such as `memmove`, `strcpy`, and `malloc`. These user-space wrappers check incoming pointer bounds and ensure that the correct bounds are associated with the returned pointers. They are also responsible for updating the BD and BTs (e.g., `memcopy` must duplicate any BD and BT entries associated with the copied memory). However, these user-space wrappers also cannot be used in the kernel because the kernel implements its own standard library.

MPXK includes new code to initialize the MPX hardware during the kernel boot process. This is done by writing the MPX configuration into each CPU's `bndcfgs` Machine Specific Register (MSR). Although MPXK does not use a BD, it still reserves an address space for the BD but does not back this up with physical memory. This is done to ensure that any erroneous and/or malicious invocations of the `bndldx` or `bndstx` instructions will cause page faults and crash the kernel instead of potentially overwriting arbitrary memory.

MPXK includes a new set of kernel-specific wrapper functions, which are implemented as normal in-kernel library functions, and used by our new MPXK GCC-plugin. The MPXK wrappers are substantially less complex (and thus easier to audit for security) than their user-space counterparts, because the MPXK wrappers do not need to include logic for updating the DB or BTs.

5.3 Binary compatibility (Mixed code)

MPXK, like MPX, is binary compatible and can therefore be used in mixed environments with both MPXK enabled code and legacy (non-instrumented) code. A fundamental problem for any binary-compatible bounds checking scheme is that the instrumentation *cannot track pointer manipulation performed by legacy code* and therefore cannot make any assumptions about the pointer bounds after the execution flow returns from the legacy code. MPX offers a partial solution by storing the pointer's value together with its bounds (using the `bndstx` instruction) before entering legacy code. When the legacy code returns and `bndldx` is used to load the bounds again, this instruction will reset the pointer bounds, i.e. making them essentially infinite, if it detects that the current pointer value has been changed. However, MPXK does not use the `bndstx` and `bndldx` instructions but instead always attempt to load such bounds using `mpxk_load_bounds`. Note that this does give an additional advantage compare to MPX, since MPXK is able to load the bounds based on the pointer value.

Function arguments, both in MPX and MPXK, rely on the caller to supply bounds. As such, these schemes cannot determine bounds for arguments when called from non-instrumented code. In both MPX and MPXK, these bounds therefore cannot be checked. As future work, we are investigating how to determine such bounds using the new MPXK bound load function described above.

5.4 Kernel instrumentation

The MPXK instrumentation is based on the existing MPX support in GCC, but uses a new GCC plugin to adapt this for use in the kernel. As described above, this new plugin instruments the kernel code with the new MPXK bound loading function, MPXK initialization code, and kernel-space wrapper functions. This plugin uses the GCC plugin system that has been incorporated into Kbuild, the Linux build system, since Linux v4.8. This means that MPXK is seamlessly integrated with the regular kernel build workflow. A developer simply needs to add predefined MPXK flags to any Makefile entries in order to include MPXK instrumentation. The plugin itself is implemented in four compiler passes, of which the first three operate on the high-level intermediate representation, GIMPLE, and the last on the lower-level RTL, as follows:

- *mpxk_pass_wrappers* replaces specific memory-altering function calls with their corresponding MPXK wrapper functions, e.g., replacing `kmalloc` calls `mpxk_wrapper_kmalloc` calls.
- *mpxk_pass_cfun_args* inserts MPXK bound loads for function arguments where bounds are not passed via the four `bndx` registers. This naturally happens when more than four bounds are passed, or due to implementation specifics for any argument beyond the sixth.
- *mpxk_pass_bnd_store* replaces `bndldx` calls with MPXK bound loads, and removes `bndstx` calls. This covers all high-level (GIMPLE) loads and saves, including return values to legacy function calls.
- *mpxk_pass_sweeper* is a final low-level pass that removes any remaining `bndldx` and `bndstx` instructions. This pass is required to remove instructions that are inserted during the expansion from GIMPLE to RTL.

6 EVALUATION

We evaluate our proposed solutions against the requirements defined in Section 3.

6.1 Security guarantees

We analyse the security guarantees of both `refcount_t` and MPXK, first through a principled theoretical analysis, and second by considering the mitigation of real-world vulnerabilities.

Reference counter overflows. With the exception of `refcount_set` and `refcount_read`, all functions that modify `refcount_t` can be grouped into *increasing* and *decreasing* functions. All increasing functions maintain the invariants that i) **the resulting value will not be smaller than the original** and that ii) **a value of zero will not be increased**. The decreasing functions maintain corresponding invariants that i) **the resulting value will not be larger than the original** and that ii) **a value of** $UINT\_MAX$ **will not be decreased**.

For example, the increasing function `refcount_add_not_zero` (Algorithm 2) maintains the invariants as follows:

- **input** = 0: Lines 3- 4 prevent the counter being increased.
- **input** = $UINT\_MAX$: Lines 6- 7 ensure the counter will never overflow.
- **input** $\in (0, UINT\_MAX)$: Lines 10- 11 ensure that the counter value cannot overflow as a result of addition.

Line 13 ensures that the addition is performed atomically, thus preventing unintended effects if interleaved threads update the counter concurrently. Regardless of how the algorithm exits, the

invariant is maintained. The same exhaustive case-by-case enumeration can be used to prove that all other `refcount_t` functions maintain the invariants.

An attacker could still attempt to cause a use-after-free error by finding and invoking an extra decrement (i.e., decrement without a corresponding increment). This is a fundamental issue inherent in all reference counting schemes. However, the errors caused by the extra decrement would almost certainly be detected early in development or testing. In contrast, missing decrements are very hard to detect through testing as they may require millions of increments to a single counter before resulting in observable errors. Thanks to the new `refcount_t`, missing decrements can no longer cause reference counter overflows.

In terms of real-world impact, `refcount_t` would have prevented several past exploits, including CVE-2014-2851, CVE-2016-0728, and CVE-2016-4558. Although it is hard to quantify the current (and future) security impact this will have on the kernel, observations during our conversion efforts support the intuition that the strict `refcount_t` API discourages unsafe implementations. For example, at least two new reference counting bugs[5] were noticed and fixed due to their incompatibility with the new API.

Out-of-bounds memory access. The objective of MPXK is to prevent spatial memory errors by performing pointer bounds checking. Specifically, for objects with known bounds, MPXK will ensure that pointers to those objects cannot be dereferenced outside the object's bounds (e.g., as would be the case in a classic buffer overflow). A fundamental limitation of bounds checking schemes is that there are various cases in which the correct object bounds cannot be (feasibly) known by the scheme. We enumerate each of these cases and show that MPXK is at least as secure as existing schemes.

Pointer manipulation: If the attacker can corrupt a pointer's value to point to a different object *without dereferencing* the pointer, this can be used to subvert bounds checking schemes. For example, object-centric schemes such as KASAN enforce bounds based on the pointer's value. If this value is changed to point within another object's bounds, the checks will be made (incorrectly) against the latter object's bounds. In theory, pointer-centric schemes such as user-space MPX should not be vulnerable to this type of attack, since they do not derive bounds based on the pointer's value. However, presumably for compatibility reasons, if MPX detects that a pointer's value has changed, it *resets* the pointer's bounds (i.e., allows it to access the full memory space). MPXK, like KASAN, will use the corrupted value to infer an incorrect set of bounds. This is therefore no worse than MPX or object-centric schemes. However, it must be noted that this type of attack requires the attacker to have a *prior exploit* to corrupt the pointer in the first place.

Legacy code: Binary-compatibility is a fundamental problem for any scheme that tracks pointers' bounds. This is manifest in two cases: i) pointers returned from legacy (non-instrumented) code to an instrumented caller, and ii) pointers passed from legacy code as arguments to an instrumented function. In both cases, the pointer bounds are not known and thus cannot be tracked by the instrumented code. These issues often cannot be addressed in binary-compatible systems, MPXK however, in case i), uses its `mpxk_load_bounds` function to determine and load the bounds.

[5] http://lkml.org/lkml/2017/6/27/409, http://lkml.org/lkml/2017/3/28/383

Table 1: CPU load measurements (in cycles).

Function	SkyLake	Sandy Bridge	Ivy Bridge-EP
atomic_inc()	15	13	10
refcount_inc()	31	37	31

One potential solution for case ii) is to add meta-data to track the bounds from legacy callers. This could be done by relaxing the strict binary-compatibility requirement, which may be feasible for the kernel, since the whole code base is typically compiled at once. The compiler could perform kernel-wide analysis while still only adding instrumentation to the intended subsystems. We plan to investigate this as future work.

One current limitation of MPXK is that the `mpxk_load_bounds` function currently can only retrieve the bounds of objects allocated by `kmalloc`. As future work, we plan to extend this to include pointers into static memory or the stack, which are not associated with any allocation pool. Even if MPXK is unable to determine the precise bounds, these pointers could still be restricted to sensible memory areas (e.g., specific stack frames).

As a practical demonstration of MPXK real-world effectiveness, we have tested MPXK against an exploit built around the recent CVE-2017-7184. The vulnerability, which affects the IP packet transformation framework `xfrm`, is a classic buffer overflow caused by omission of an input size verification. We first confirmed that we can successfully gain root privileges on a current Ubuntu 16.10 installation running a custom-built v4.8 kernel using the default Ubuntu kernel configuration. We then recompiled the kernel applying MPXK on the `xfrm` subsystem, which caused the exploit to fail with a bound violation reported by MPXK.

6.2 Performance

Reference counter overflows. Although the `refcount_t` functions consist mainly of low-overhead operations (e.g., additions and subtractions), they are often used in performance-sensitive contexts. Therefore, while absolute overhead of individual calls can be illuminating, we need to estimate the practical impact on overall performance. We performed various micro-benchmarks of the individual functions during the `refcount_t` development[6]. As shown in Table 1, `refcount_inc` introduces an average overhead of 20 CPU cycles, compared to `atomic_inc`. However micro-benchmarks cannot be considered in isolation when evaluating the overall performance impact.

To gauge the overall performance impact of `refcount_t` we conducted extensive measurements on the networking subsystem using the Netperf [31] performance measurement suite. We chose this subsystem because i) it is known to be performance-sensitive; ii) it has a standardized performance measurement test suite; and iii) we encountered severe resistance on performance grounds when proposing to convert this subsystem to use `refcount_t`. The concerns are well-founded due to the heavy use of reference counters (e.g., when sharing networking sockets and data) under potentially substantial network loads. The main challenge when evaluating performance impact on the networking subsystem is that there are no standardized workloads, and no agreed criteria as to what

[6] http://lwn.net/Articles/718280/

Table 2: Netperf refcount usage measurements

Netperf Test type	base	refcount	change (stddev)
UDP CPU use (%)	0.53	0.75	+42.1% (34%)
TCP CPU use (%)	1.13	1.28	+13.3% (3%)
TCP throughput (MB/s)	9358	9305	-0.6% (0%)
TCP throughput (tps)	14909	14761	-1.0% (0%)

constitutes "acceptable" performance overhead. Our test setup consisted of physically connected server and client machines running the mainline `v4.11-rc8` kernel. We measured the real-world performance impact of converting all 78 reference counters in the networking subsystem and networking drivers from `atomic_t` to `refcount_t`.

As shown in Table 2, our Netperf measurements include CPU utilization for UDP and TCP streams, and TCP throughput in terms of MB/s and transactions per second (tps). The results indicate that throughput loss is negligible, but the average processing overhead can be substantial, ranging from 13% for UDP to 42% for TCP. The processing overhead can in many situations be acceptable; Desktop systems typically only use networking sporadically, and when they do, the performance is typically limited by the ISP link speed, not CPU bottlenecks. In contrast, this might not be the case in servers or embedded systems, and routers in particular, where processing resources may be limited and networking a major contributor to system load. However, such systems are typically closed systems, i.e., it is not possible to install additional applications or other untrusted software, so therefore their attack surface is already reduced. These considerations are the reason for the `CONFIG_REFCOUNT_FULL` kernel configuration option, which can be used to disable the protection offered by `refcount_t` when the performance overhead is too high, whilst still allowing all other systems to benefit from these new protection mechanisms (which are expected to be enabled by default in the future).

Out-of-bounds memory access. MPXK incurs three types of performance overhead (excluding compile-time overheads). First, the instrumentation naturally increases CPU utilization and kernel size. Although we do not use the costly MPX bound storage, some memory will nonetheless be used to store static global and intermediary bounds. Memory use comparisons, however, indicate that the memory overhead is negligible. When deploying MPXK over the `xfrm` subsystem, the memory overhead for in-memory kernel code is $110KB$, which is a 0.7% increase in total size. Kernel image size overhead is increased by $45KB$ or 0.6%. Some CPU overhead is expected due to the instrumentation and bound handling. To measure this, we conducted micro-benchmarks on `kmalloc` and `memcpy`, comparing the performance of MPXK and KASAN, with the results shown in Table 3.

To better gauge the actual runtime performance impact, we again applied MPXK to `xfrm` and measured the impact on an IPSec tunnel where `xfrm` manages the IP package transformations. For measurements we used Netperf, running five-minute tests for a total of 10 iterations. The test system was running Ubuntu 16.04 LTS with a v4.8 Linux kernel. As shown in Table 4, there is a small reduction in TCP throughput, but the impact on CPU performance is negligible.

The micro-benchmarks indicate that MPXK, as expected, introduces a measurable performance overhead. However, compared to KASAN, the performance overhead is relatively small. The large scale tests on `xfrm` and Netperf confirm that similar performance measurements hold for full real-world systems. It should further be noted that MPXK is specifically designed for modular deployment, thus ensuring that it can be deployed incrementally without affecting performance sensitive modules or subsystems.

6.3 Deployability

Reference counter overflows. Deploying a new data type into a widely used system such as the Linux kernel is a significant real-world challenge. From a usability standpoint, the API should be as simple, focused, and self-documenting as reasonably possible. The `refcount_t` API in principle fulfils these requirements by providing a tightly focused API with only 14 functions. This is a significant improvement over the 100 functions provided by the `atomic_t` type previously used for reference counting. Of the 233 patches we submitted, 123 are currently accepted, and it is anticipated that the rest will be accepted in the near future. The `refcount_t` type has also been taken into use in independent work by other developers[7]. This gives a strong indication that the usability goals are met in practice. Our contributions also include the Coccinelle pattern used for analysis and detection of potential reference counting schemes that can be converted to `refcount_t`. This pattern is currently in the process of being contributed to the mainline kernel, at which point it can be used by individual developers and in various places in the Linux kernel testing infrastructure.

Out-of-bounds memory access. As with MPX, our MPXK design considers usability as a primary design objective. It is binary compatible, meaning that it can be enabled for individual translation units. It is also source-compatible in that it does not require any changes to source code. In some cases, pointer bounds checking can interfere with valid pointer arithmetic or other atypical pointer use sometimes seen in high-performance implementations. However, such compatibility issues are usually only present in architecture-specific implementations of higher-level APIs and can thus be accounted for in a centralized manner by annotating incompatible functions to prevent their instrumentation. Compatibility issues are also usually found during compile time and are thus easily detected in development. MPXK is fully integrated into Kbuild, providing predefined compilation flags for easy deployment. Using the `MPXK_AUDIT` parameter, MPXK can also be configured to run in permissive mode where violations are only logged, which is useful for development and incremental deployment.

The MPXK code-base is largely self-contained and thus easy to maintain. The only exceptions are the wrapper functions that are used to instrument memory manipulating functions. However, this is not a major concern because the kernel memory managing and string API is quite stable and changes are infrequent. The instrumentation is all contained in a GCC-plugin and is thus not directly dependent on GCC internals. MPXK is thus both easily deployable and easy to maintain.

[7] http://lkml.org/lkml/2017/6/1/762

Table 3: MPXK and KASAN CPU overhead comparison.

	baseline	KASAN		MPXK	
	time (stddev)	*ns* diff (stddev)	% diff	*ns* diff (stdev)	% diff
No bound load.					
memcpy, 256 B	45 (0.9)	+85 (1.0)	+190%	+11 (1.2)	+30%
memcpy, 65 kB	2340 (4.3)	+2673 (58.1)	+114%	+405 (5.1)	+17%
Bound load needed.					
memcpy, 256 B	45 (0.8)	+87 (0.9)	+195%	+70 (1.5)	+155%
memcpy, 65 kB	2332 (5.8)	+2833 (28.2)	+121%	+475 (15.0)	+20%

Table 4: Netperf measurements over an IPSec tunnel with the `xfrm` subsystem protected by MPXK.

Netperf test	baseline	MPXK	change (stddev)
UDP CPU use (%)	24.97	24.97	0.00% (0.02)%
TCP CPU use (%)	25.07	25.15	0.31% (0.29)%
TCP throughput (MB/s)	646.69	617.95	-4.44% (4.61)%
TCP throughput (tps)	1586.79	1547.85	-2.45% (1.66)%

7 RELATED WORK

Research on memory errors, both temporal and spatial, has long roots both in the industry and in academia. Many solutions have been presented to eliminate, mitigate and detect memory errors of various types, but they typically exhibit characteristics that prevent their wide deployment. Purify [16], Shadow Guarding [33, 34], SoftBound [26], as well as approaches in [18], [29, 47, 49], and [12], have non-acceptable run-time performance. CCured [28] and Cyclone [14] required source code changes. In addition CCured [28], Cyclone [14] and approach [3] are not backward compatible. Moreover, these solutions have typically focused on user-space and have not been intended or even supported in kernel-space. Some recent notable exceptions are kCFI [40] and KENALI [44], but they have not from an implementation standpoint targeted actual mainline kernel adoption. To our knowledge, none of these systems are in production use as run-time security mechanisms. A notable exception is the PaX/Grsecurity patches that have pioneered many in-use security mechanisms such as Address Space Layout Randomization [35]. PaX/Grsecurity also includes a `PAX_REFCOUNT` [5] feature that prevents reference counter overflows, but this requires extensive modification of the underlying atomic types. These extensive changes, and a potential race condition, made this feature unsuitable for mainline kernel adoption [5].

Some tools have reached high prominence and active use in the debugging of memory errors. These include Valgrind [30] which offers a suite of six tools for debugging, profiling, and error detection, including the detection of memory errors. AddressSanitizer [41] provides something of a gold standard in run-time memory error detection, but is unsuitable for run-time production use due to performance overhead. KASAN [19] is integrated into the mainline Linux kernel and offers similar protections for the kernel, but again incurs performance overheads unsuitable for most production use cases.

From a production perspective, much work has focused on preventing the exploitation of memory errors. Exploitability of buffer overflows, whether stack-based or heap-based, can be limited by preventing an attacker from misusing overflown data. One early mitigation technique is the non-executable (NX) bit for stack, heap and data pages [36, 43]. However, this can be circumvented by using overflows for execution redirection into other legitimate code memory, such as the C library in so called return-to-libc attacks, or more generally to any executable process memory in return oriented programming attacks [20]. Another mitigation technique is the use stack canaries that allow the detection of overflows, e.g., StackGuard [10] and StackShield [45]. Such detection techniques can typically be circumvented or avoided using more selective overflows that avoid the canaries, or by exploiting other callback pointers such as exception handlers. Probabilistic mitigation techniques such as memory randomization [35, 46] are commonly used, but have proven difficult to secure against indirect and direct memory leaks that divulge randomization patterns, not to mention later techniques such as heap spraying [39].

In contrast to the development/debugging temporal safety tools and the run-time-friendly mitigation measures, MPXK is a run-time efficient system focused on the prevention of the underlying memory errors. MPXK is conceptually similar to previous solutions such as [14, 22, 23, 26, 28]. Like these systems, MPXK does not use fat-pointers, which alter the implementation of pointers to store both the pointer and the bounds together, but instead preserves the original memory layout. Unlike purely software-based systems such as SoftBound [26], MPXK has the advantage of hardware registers for propagating bounds and hardware instructions for enforcing bounds. HardBound [11] employs hardware support similar to MPXK but has a worst case memory overhead of almost 200% and has only been simulated on the micro-operation level and to date lacks any existing hardware support. Unlike MPXK, none of these previous schemes have been designed for use in the kernel.

8 DISCUSSION

The solutions proposed in this paper and related Linux kernel mainline patches are a step towards better memory safety in the mainline Linux kernel. One sobering fact when working with production systems and distributed development communities, such as the Linux kernel, is that there is always a trade-off between security and deployability. Security solutions outside the mainline kernel, such as the long-lived PaX/Grsecurity [15] patches can thrive, but in order to provide wide-spread security to a diverse space of devices, it is imperative to achieve integration into the mainline kernel.

Future work:

Our efforts to convert the remaining reference counters to `refcount_t` continue, but the initial work has already sparked other implementation efforts and a renewed focus on the problem. There are active efforts to migrate to the mainline kernel solutions that provide high-performance architecture-independent implementations [2].

Several MPXK improvements are also left for future work. Our support for bound loading can be extended with support for other allocators and memory region-based bounds for pointers not dynamically allocated. More invasive instrumentation could also be used to improve corner cases where function calls require bound loading. The wrapper implementations could similarly be improved by more invasive instrumentation, namely by forgoing wrappers altogether and instead employing direct instrumentation at call sites. This approach is not feasible on vanilla MPX due complications caused by the bound storage, but would be quite reasonable for MPXK.

MPX: suggestions for improvements:

While working with Intel MPX and adapting it for kernel use, we have identified some aspects that limit its usability for this use in particular, but also in more general use cases. One particular challenge was lacking documentation of both hardware and compiler behavior in atypical cases. In many cases, such behavior is impossible to determine without either manual run-time testing or extensive compiler source code analysis. While these limitations might not affect trivial user-space applications, they are glaring problems when working with performance-critical low-level systems such as the Linux kernel. This is not ideal for a technology that aims for wide adoption and can easily become an insurmountable barrier. From a pure hardware perspective, MPX currently provides only four registers for storing bounds. By increasing the register count, the instrumentation would be more efficient overall, and in particular function argument bound propagation would benefit by being able to omit costly bound loads.

Working with Linux kernel community:

The path to successful non-trivial mainline kernel patches is often a long and rocky road, especially for security-related features that have performance implications. Based on our experience in working with the Linux kernel developers in this project, we offer the following guidance for other researchers who also aim to have their solutions merged into the mainline Linux kernel.

Understanding context. It is not enough to simply read the Linux kernel contribution guidelines [24] and follow them when developing your proposal. It is much more useful to understand the *context* of the subsystem to which you are attempting to contribute. By context, we mean the recent history of the subsystem and planned developments, standard ways in which it is verified and tested, and the overall direction set by its maintainers. Presenting your contribution by embedding it in this context will increase the chances of its getting a fair hearing. For example, in our MPXK work, instead of contributing changes directly to the GCC core, we implemented our GCC instrumentation as a standalone GCC plugin (Section 5) using the GCC kernel plugin framework that was just recently added to the mainline Linux kernel. This allowed us to be aligned with the overall direction of the GCC compiler development.

Fine-grained configurability. The Linux kernel runs on very different types of devices and environments with different threat models and security requirements. Thus any security solution (and especially those that have performance or usability implications) must not be monolithic but support the ability to be configured to several different *grades*. For example, we designed MPXK so that its level of protection can be independently enabled on each compilation unit or subsystem to protect places where it is most needed (see Section 6.3). Similarly, in our reference counter protection work, in addition to having a configuration flag for turning off the protection behind the `refcount_t` interface, there is ongoing work to provide an architecture-specific fast assembly implementation with only slightly relaxed security guarantees [2]. Also, the conversion to the new `refcount_t` API can happen independently for each reference counter. Kernel maintainers can thus gradually change their code to use `refcount_t` rather than requiring all kernel code to be changed at once (in fact, the latter is a major reason for kernel developers turning down the PaX/Grsecurity solution for reference counters.).

Timing. Developing and deploying any new feature that affects more than a single kernel subsystem takes considerable time. This is due to the number of different people involved in maintaining various kernel subsystems and the absence of strict organization of the development process. Researchers should plan for this to make sure enough time is allocated. Also, one has to take into account various stages of kernel release process to understand when it is the best time to send your patches for review and get feedback from maintainers. For example, it took us a full year to reach the current state of reference counter protection work and have 123 out of 233 patches merged.

Finally, even if a proposed feature is not accepted in the end, both maintainers and security researchers can learn from the process, which can eventually lead to better security in the mainline kernel.

9 CONCLUSION

Securing the mainline Linux kernel is a vast and challenging task. In this paper we present a set of solutions that on the one hand limit the exploitability of memory errors by eliminating reference counter overflows and on the other provide hardware assisted solution for enforcing pointer bounds. Both solutions are aimed at practical deployability, with reference counter protection in particular already being widely deployed in the mainline kernel. While our solutions arguably exhibit some limitations, they nonetheless strike a pragmatic balance between deployability and security, thus ensuring they are in a position to benefit the millions of devices based on the mainline kernel.

REFERENCES

[1] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. 2009. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors.. In *USENIX Security Symposium*. 51–66.
[2] Anonymized. 2017. Anonymized. (2017).
[3] Todd M. Austin, Scott E. Breach, Gurindar S. Sohi, Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. 1994. Efficient detection of all pointer and array access errors. *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation - PLDI '94* 29, 6 (1994), 290–301. https://doi.org/10.1145/178243.178446

[4] Mick Bauer. 2006. Paranoid penguin: an introduction to Novell AppArmor. *Linux Journal* 2006, 148 (2006), 13.

[5] Rodrigo Branco. 2015. Grsecurity forum — Guest Blog by Rodrigo Branco: PAX_REFCOUNT Documentation. (2015). https://forums.grsecurity.net/viewtopic.php?f=7&t=4173

[6] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M Frans Kaashoek. 2011. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*. ACM, 5.

[7] Coccinelle 2017. Coccinelle Project. http://coccinelle.lip6.fr/. (2017).

[8] George E. Collins. 1960. A Method for Overlapping and Erasure of Lists. *Commun. ACM* 3, 12 (1960), 655–657. https://doi.org/10.1145/367487.367501

[9] Kees Cook. 2016. Status of the Kernel Self Protection Project. *www.outflux.net/slides/2016/lss/kspp.pdf* (2016).

[10] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. 1998. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks.. In *USENIX Security Symposium*, Vol. 98. 63–78.

[11] Joe Devietti, Colin Blundell, Milo MK Martin, and Steve Zdancewic. 2008. Hardbound: architectural support for spatial safety of the C programming language. In *ACM SIGARCH Computer Architecture News*, Vol. 36. ACM, 103–114.

[12] Dinakar Dhurjati and Vikram Adve. 2006. Backwards-compatible array bounds checking for C with very low overhead. In *Proceedings of the 28th international conference on Software engineering*. ACM, 162–171.

[13] dm-verify 2017. dm-verity project pages. http://source.android.com/security/verifiedboot/dm-verity. (2017).

[14] Dan Grossman, Michael Hicks, Trevor Jim, and Greg Morrisett. 2005. Cyclone: A type-safe dialect of C. *C/C++ Users Journal* 23, 1 (2005), 112–139.

[15] Grsecurity 2017. grsecurity. https://grsecurity.net. (2017).

[16] Reed Hastings and Bob Joyce. 1991. Purify: Fast detection of memory leaks and access errors. In *In Proc. of the Winter 1992 USENIX Conference*. 125–138.

[17] IMA 2017. Integrity Measurement Architecture (IMA) wiki pages. http://sourceforge.net/p/linux-ima/wiki/Home/. (2017).

[18] Richard WM Jones and Paul HJ Kelly. 1997. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the 3rd International Workshop on Automatic Debugging; 1997 (AADEBUG-97)*. Linköping University Electronic Press, 13–26.

[19] KASAN 2017. The Kernel Address Sanitizer (KASAN). www.kernel.org/doc/html/v4.10/dev-tools/kasan.html. (2017).

[20] Sebastian Krahmer. 2005. X86-64 Buffer Overflow Exploits and the Borrowed Code Chunks Exploitation Technique. (2005), 1–20. https://trailofbits.github.io/ctf/exploits/references/no-nx.pdf

[21] KSPP 2017. Kernel Self Protection Project wiki. http://www.kernsec.org/wiki/index.php/Kernel$_{S}$elf$_{P}$rotection$_{P}$roject. (2017).

[22] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2017. SGXBOUNDS: Memory Safety for Shielded Execution. In *Proceedings of the 2017 ACM European Conference on Computer Systems (EuroSys)*.

[23] Albert Kwon, Udit Dhawan, Jonathan M Smith, Thomas F Knight Jr, and Andre DeHon. 2013. Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 721–732.

[24] Linux Kernel Documentation 2017. Submitting patches: the essential guide to getting your code into the kernel. https://www.kernel.org/doc/html/v4.12/process/submitting-patches.html. (2017).

[25] Paul E McKenney. 2007. *Overview of linux-kernel reference counting*. Tech. Rep. N2167=07-0027. Linux Technology Center, IBM Beaverton. http://www.open-std.org/JTC1/sc22/wg21/docs/papers/2007/n2167.pdf

[26] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2009. SoftBound: Highly compatible and complete spatial memory safety for C. 44, 6 (2009), 245–258.

[27] National Vulnerability Database. 2017. NVD: Statistics. (2017). https://nvd.nist.gov/vuln/search/statistics?adv$_{s}$earch=true&form$_{t}$ype=advanced&results$_{t}$ype=statistics&query=kernel

[28] George C Necula, Scott McPeak, and Westley Weimer. 2002. CCured: Type-safe retrofitting of legacy code. In *ACM SIGPLAN Notices*, Vol. 37. ACM, 128–139.

[29] Nicholas Nethercote and Jeremy Fitzhardinge. 2004. Bounds-Checking Entire Programs without Recompiling. *Proceedings of the Second Workshop on Semantics Program Analysis and Computing Environments for Memory Management SPACE 2004* (2004). http://www.njn.valgrind.org/pubs/bounds-checking2004.ps

[30] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Vol. 42. ACM, 89–100.

[31] Netperf 2017. Netperf Project. http://hewlettpackard.github.io/netperf. (2017).

[32] Vitaly Nikolenko. 2016. Exploiting COF Vulnerabilities in the Linux kernel. *ruxcon.org.au/assets/2016/slides/ruxcon2016-Vitaly.pdf* (2016).

[33] Harish Patil and Charles Fischer. 1997. Low-cost, concurrent checking of pointer and array accesses in C programs. *Softw., Pract. Exper.* 27, 1 (1997), 87–110.

[34] Harish Patil and Charles N Fischer. 1995. Efficient Run-time Monitoring Using Shadow Processing.. In *AADEBUG*, Vol. 95. 1–14.

[35] PaX Team. 2003. PaX address space layout randomization (ASLR). (2003). http://pax.grsecurity.net/docs/aslr.txt

[36] PaX Team. 2003. PaX non-executable pages design & implementation. (2003). http://pax.grsecurity.net

[37] Supriya Raheja, Geetika Munjal, et al. 2016. Analysis of Linux Kernel Vulnerabilities. *Indian Journal of Science and Technology* 9, 48 (2016).

[38] Ramu Ramakesavan, Dan Zimmerman, and Pavithra Singaravelu. 2015. Intel Memory Protection Extensions (Intel MPX) enabling guide. (2015). http://pdfs.semanticscholar.org/bd11/4878c6471cb5ae28546a594bf25ba5c25c6f.pdf

[39] Paruj Ratanaworabhan, V Benjamin Livshits, and Benjamin G Zorn. 2009. NOZZLE: A Defense Against Heap-spraying Code Injection Attacks.. In *USENIX Security Symposium*. 169–186.

[40] Sandro Rigo, Michalis Polychronakis, and Vasileios P Kemerlis. 2017. DROP THE ROP Fine-grained Control-flow Integrity for the Linux Kernel. https://www.blackhat.com/docs/asia-17/materials/asia-17-Moreira-Drop-The-Rop-Fine-Grained-Control-Flow-Integrity-For-The-Linux-Kernel-wp.pdf. (2017).

[41] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker.. In *USENIX Annual Technical Conference*. 309–318.

[42] Stephen Smalley, Chris Vance, and Wayne Salamon. 2001. Implementing SELinux as a Linux security module. *NAI Labs Report* 1, 43 (2001), 139.

[43] Solar Designer. 1997. Linux kernel patch to remove stack exec permission. (1997). http://seclists.org/bugtraq/1997/Apr/31

[44] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William R. Harris, Taesoo Kim, and Wenke Lee. 2016. Enforcing Kernel Security Invariants with Data Flow Integrity. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[45] StackShield. 2011. A stack smashing technique protection tool for Linux. (2011). http://www.angelfire.com/sk/stackshield

[46] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K Iyer. 2003. Transparent runtime randomization for security. In *Reliable Distributed Systems, 2003. Proceedings. 22nd International Symposium on*. IEEE, 260–269.

[47] Wei Xu, Daniel C DuVarney, and R Sekar. 2004. An Efficient and Backwards-compatible Transformation to Ensure Memory Safety of C Programs. *SIGSOFT Softw. Eng. Notes* 29, 6 (2004), 117–126. https://doi.org/10.1145/1041685.1029913

[48] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. 2015. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 414–425.

[49] Suan Hsi Yong and Susan Horwitz. 2003. Protecting C programs from attacks via invalid pointer dereferences. In *ACM SIGSOFT Software Engineering Notes*, Vol. 28. ACM, 307–316.

APPENDIX

Listing 3: Coccinelle pattern for finding reference counters in the Linux kernel

```
// Check if refcount_t type and API should be used
// instead of atomic_t type when dealing with refcounters
// Confidence: Moderate
// URL: http://coccinelle.lip6.fr/
// Options: --include-headers
virtual report
@r1 exists@
identifier a, x;
position p1, p2;
identifier fname =~ ".*free.*";
identifier fname2 =~ ".*destroy.*";
identifier fname3 =~ ".*del.*";
identifier fname4 =~ ".*queue_work.*";
identifier fname5 =~ ".*schedule_work.*";
identifier fname6 =~ ".*call_rcu.*";
@@
(
 atomic_dec_and_test@p1(&(a)->x)          |
 atomic_dec_and_lock@p1(&(a)->x, ...)     |
 atomic_long_dec_and_lock@p1(&(a)->x, ...)|
 atomic_long_dec_and_test@p1(&(a)->x)     |
```

```
 atomic64_dec_and_test@p1(&(a)->x)        |
 local_dec_and_test@p1(&(a)->x)
)
...
(
 fname@p2(a, ...);  |
 fname2@p2(...);    |
 fname3@p2(...);    |
 fname4@p2(...);    |
 fname5@p2(...);    |
 fname6@p2(...);
)
@script:python depends on report@
p1 << r1.p1;
p2 << r1.p2;
@@
msg = "atomic_dec_and_test␣variation
␣␣␣␣␣␣before␣object␣free␣at␣line␣\%s."
coccilib.report.print_report(p1[0],
                  msg \% (p2[0].line))
@r4 exists@
identifier a, x, y;
position p1, p2;
identifier fname =~ ".*free.*";
@@
(
 atomic_dec_and_test@p1(&(a)->x)          |
 atomic_dec_and_lock@p1(&(a)->x, ...)     |
 atomic_long_dec_and_lock@p1(&(a)->x, ...)|
 atomic_long_dec_and_test@p1(&(a)->x)     |
 atomic64_dec_and_test@p1(&(a)->x)        |
 local_dec_and_test@p1(&(a)->x)
)
...
y=a
...
fname@p2(y, ...);
@script:python depends on report@
p1 << r4.p1;
p2 << r4.p2;
@@
msg = "atomic_dec_and_test␣variation
␣␣␣␣␣␣before␣object␣free␣at␣line␣\%s."
coccilib.report.print_report(p1[0],
                  msg \% (p2[0].line))
@r2 exists@
identifier a, x;
position p1;
@@
(
atomic_add_unless(&(a)->x,-1,1)@p1      |
atomic_long_add_unless(&(a)->x,-1,1)@p1 |
atomic64_add_unless(&(a)->x,-1,1)@p1
)
@script:python depends on report@
p1 << r2.p1;
@
msg = "atomic_add_unless"
coccilib.report.print_report(p1[0], msg)
@r3 exists@
identifier x;
position p1;
@@
(
x = atomic_add_return@p1(-1, ...);       |
x = atomic_long_add_return@p1(-1, ...); |
x = atomic64_add_return@p1(-1, ...);
)
@script:python depends on report@
p1 << r3.p1;
@@
msg = "x␣=␣atomic_add_return(-1,␣...)"
coccilib.report.print_report(p1[0], msg)
```