Preventing Privilege Escalation

Final Report – Group 5

Group Members: CXM1010, DXD946, KNM986, MMA976, GTB984, AXX945

1. Introduction

Privilege escalation is the exploitation of software vulnerabilities that results in elevated and unauthorized access to resources which are normally protected from applications or specific users. This is an important component of many cyber-attacks. If an adversary manages to exploit privileges and gain unauthorised permissions, they can potentially gain full control over a system and its processes.

The primary focus of this report is to dissect Preventing Privilege Escalation (Provos *et al*) with respect to its contribution to the field since its release in 2003. The paper itself focuses on reviewing the security primitive of privilege separation - a security measure that is used to help prevent privilege escalation - as a realisation of the principle of least privilege. We will then discuss the dangers presented by privilege escalation attacks and explore several concepts and techniques using in their prevention.

The primary contribution of Provos et al is a system of privilege separation based on a monitor-slave model in conjunction with a secure and rigidly defined interface. Their aim was to reduce the amount of code with elevated privileges such that a vulnerability is more likely located within unprivileged code. They examine OpenSSH as an implementation of this model.

This report first outlines a literature review that discusses related previous work connected to the principle of least privilege and privilege separation. Then, it evaluates the result of three different practical studies. Finally, it concludes with a summary of our critical analysis.

Our scripts, and some more detailed results, are available at https://github.com/CyberSecDump/SecureProgramming

2. Literature Review

Provos *et al* was published in 2003 during the USENIX Security Symposium. In the last few years this symposium has been considered one of the best in the field, with a “CORE 2018 Rating” of “A*” and a “Guide2Research Overall Ranking” of “13” (Guide2Research, 2019). Records of the Symposium during the year of the paper's publication could not be found but considering that thirteen previous symposiums had already taken place at that time and coupled with modern ratings, it is safe to assume that the venue was in good standing. The paper also has been cited 386 times since its publication, and this number has been increasing yearly (Scholar.google.com, 2019); these citations have been mostly positive, the paper is used as a technical touchstone for many investigations and implementations.

The concept of privilege separation as a mechanism for preventing privilege escalation appears to be a relatively recent development at the time our selected paper was written. Provos *et al* notes the development of Privman (Kilpatrick, 2003), a library supporting privilege separation developed in the same year. They examine in detail OpenSSH as a model of privilege separation, but this product only achieved privilege separation in 2002, the year before the paper was published. We found no earlier references to this concept.

Provos et al notes the use of several established approaches to security, including type-safe languages, protection domains, and “application confinement”, a general term that includes jailing and sandboxing. There are numerous previous works exploring these approaches. They cite a summary by *Gong et al* (1997), which explores the use of privilege domains and sandboxing in JDK 1.2, and they note that the principle of least privilege was being discussed at least as early as 1974 (John H Saltzer). Similarly, the concept of jailing has existed at least since the release of chroot in 1979. However, the paper explicitly notes that these approaches can be restrictive and platform-specific, while they aim to explore a more generic and widely applicable approach. As such, we consider their examination of the emerging concept of privilege separation to be an important development.

2.1 Principle of Least Privilege

The principle of least privilege is the foundation for privilege separation. This principle states that a user or process should only have privileges (and thus access to resources) which are necessary for it to perform intended functions, and that following the principle limits both the possibility of damage to data and the occurrence of unintended behaviour. Provos *et al* explicitly agree with this sentiment and identify the principle as key motivation for their work. Application of the Principle provides several advantages:

- **Stability:** Code that makes fewer privileged operations needs less testing and is less likely to be incompatible with other applications.
- **Security:** Vulnerable code with limited privilege gives attackers less of a foothold when compromised. Likewise, limited-privilege malfunctioning code can cause less damage.
- **Deployment:** Applications that require fewer privileges are easier to deploy in their environment.

Provos *et al* primarily focus on the second of these advantages in their exploration of privilege separation. They also note the importance of the first when examining auxiliary benefits of the code separation process in code audit. We examine the third advantage when we examine modern developments later in section 2.6.

In operating systems, the kernel always runs with the highest privilege since it is the OS core and has hardware access. Processes perform privileged operations via system calls to the kernel. The paper makes use of this functionality in their monitor-slave model by only granting access to carefully selected system calls to specific areas of code, typically via a precisely controlled interface.

Complex applications, and particularly those that use third-party modules or libraries, are statistically more likely to have bugs that can grant unintended privileges. This can complicate the application of the principle of least privilege, and exploitable vulnerabilities may still go undetected. This emphasises the importance of also using other defensive programming techniques such as integrity checks and the active prevention of buffer overflow. Provos *et al* mention a specific instance of this problem in their security analysis of OpenSSH code, specifically a vulnerability in *zlib*, and they note how the dangers of this vulnerability were reduced due to *zlib's* encapsulation within code with limited privileges.

2.2 Privilege Separation

Provos *et al* presents privilege separation as a technique that “reduce[s] the amount of code that runs with special privilege without affecting or limiting the functionality of the service”. They observe in their studies of OpenSSH that only 32.3% of the codebase requires elevated privileges and note that, if programming errors are evenly distributed, two thirds will occur outside of areas of code that require those privileges. They also note that, for services that deal with user authentication (such as OpenSSH), post-authentication programming errors “[limit] the degree of escalation because the user is already authorized to hold some privilege”, whereas obtaining superuser privileges outside of the authentication process “presents a greater degree of escalation”. They present these observations as evidence that privilege separation has significant and measurable benefits to security.

Provos *et al* also states that the encapsulation of privileged operations in a separate process can improve the effectiveness of code auditing. They argue that a “source code audit can focus on code that is executed with special privilege” which can “further reduce the incidence of unauthorised privilege escalation”. While vulnerabilities in unprivileged code may go undetected, a compromised unprivileged process is far less dangerous than a privileged one. They also observe that applying privilege separation yields code that is more modular and better organised, and thus intrinsically easier to audit effectively.

Both arguments above are based on reducing the probability of an escalatory vulnerability; however, this should not be mistaken as a guarantee that no vulnerability exists in privileged code. Believing that this model alone provides a perfectly secure monitor strongly violates the principle of defence in depth.

A fundamental component of Provos *et al*’s model is secure inter-process communication (IPC). They define the communication rigidly: “A slave may request different types of privileged operations from the monitor […] the monitor checks every request to see if it is allowed". They examine OpenSSH's monitor-slave IPC and note that "at any time, the number of requests that the slave may issue are limited by the state machine [of the monitor]", as well as the type of request. They state that this "reduce[s] the attack profile available to an intruder who has compromised the slave process"; this restrictive system is impressive and severely restricts the effectiveness of polling and brute force attacks.

While Provos *et al* observes that an authorised user gaining further privileges is less dangerous than an unauthorised user gaining superuser privileges, the degree of danger is also related to the context of the system and needs to be carefully evaluated. If the escalated but otherwise authorised user is exploiting a machine that has no access to important systems or confidential information, or indeed they cannot escape their local machine at all, then it may not be worth the expenditure of resources to give special consideration to that method of escalation. However, the difference in capability between a normal user and a superuser can be vast and should not be underestimated or trivialised. Furthermore, authorised users cannot be assumed to be any more trustworthy than an unknown adversary, particularly when considering the possibility of a non-technical compromise of an authorised user's credentials. We explore this kind of escalation further into this paper, using a vulnerable version of nmap.

Provos *et al* accept that privilege separation is not universally effective. They mention that privilege separation by itself provides little protection against denial of service; a compromised slave can be used to launch a resource exhaustion attack "by forking new processes or by running very time intensive computations". We were pleased to note that they suggest enforcing resource limitations and monitoring of slave processes, a recognition of the defence in depth principle.

Provos *et al* promotes the use of jailing via `chroot` as a security mechanism; this is comparable to sandboxing. They observe that OpenSSH uses `chroot` jailing for slave processes and that this significantly reduces the operations available to a compromised slave. This is a vital component of privilege separation and prevents direct interference with the OS, hardware, or other applications as code can only access the restricted set of resources that it legitimately requires.

2.3 Privilege Separation in Operating Systems

UNIX security is first mentioned in literature by Dennis M. Ritchie (1979). He states that UNIX was not designed in a security-first manner. The area of security in which UNIX is theoretically the weakest is in protecting against crashing or at least crippling the operation of the system. This leads to security policies designed to prevent users from achieving this damage. Chen *et al* (2002) describe the approaches taken by different operating systems to handle coarse-grained control of privilege separation.

Linux introduces a capability model for finer-grained control of privileges beyond just root and non-root. It allocates multiple user and group IDs to processes and applies the concept of *effective UID*, which determines the privileges of a process at a given time and can be altered (within a system of limitations). This is a useful foundation for the implementation of privilege separation.

These features allow the partitioning of source code based on different levels of privilege, though this can be difficult, as discussed by Kilpatrick (2003). There are two approaches: *automatic privilege separation* and *manual privilege separation.*

2.4 Automatic Privilege Separation

PrivTrans (Brumley *et al*, 2004) presents a system for automatic privilege separation that uses the same model as Provos *et al* and is more comprehensive than that of the privilege separation library *Privman* (Kilpatrick, 2003), since it encompasses both access control and protecting data derived from privileged resources.

They identify that the creation of a monitor and slave from the given source code using automatic privilege separation requires three key features: A mechanism for automatically identifying privileged resources, a Remote Call Procedure (RPC) mechanism for communication between the monitor and slave, and a storage mechanism within the monitor for caching the result of privileged operations. However, they also note that automatic privilege separation has drawbacks:

- PrivTrans' automatic dataflow analysis results in code with non-optimal performance compared to that produced by a programmer with application and context-specific knowledge.
- Privman cannot precisely discern the security requirements of an application. The configuration, while more detailed than Unix system privilege, may still allow excessive privilege to an attacker.

- PrivTrans assumes that the original program accesses privileged resources through function calls; while this is true is most cases, it is not absolute.

Conversely, manual process partitioning can use developer knowledge to precisely match the policy to the application's specific needs. For example, OpenSSH is written such that some privileged requests can only happen once.

2.5 Challenges when implementing Privilege Separation

Some programs rely on behaviors that can be disrupted when implementing privilege separation. While these issues are surmountable, they can introduce additional complexities.

- Programs that already manage their own privileges may break if naïvely privilege separated.
- File descriptor numbering will be offset in monitor-slave implementations that use sockets, though poll calls can resolve this issue.
- File descriptors shared by several processes can be closed by one process while still needed by others. PrivTrans solves this issue by forking new monitors for each slave.
- Data structures may contain both privileged and unprivileged data. PrivTrans uses opaque identifiers to distinguish between privileged and unprivileged data, though this does not always resolve the issue.
- Methods of state retention for processes are platform-specific, making it difficult for processes to change their user ID. PrivTrans provides a system call that lets processes change the uid of its slaves (superusers for any process), and Provos et al explore state export via shared memory.

2.6 Modern approaches

Since the publication of Provos *et al*, developments in computer technology have led to the creation and use of entirely new software platforms. While the core concepts they discuss are still vital in the current day, these platforms did not exist at the time of writing. We felt it was essential to augment our review of the paper with recent innovations.

Mobile apps and browser extensions are two modern instances where the casual execution of potentially unsafe code is allowed or even encouraged. As such, privilege separation and associated techniques such as sandboxing have become mandatory components of such platforms. This has not always been the case; Internet Explorer's original Browser Helper Object interface permitted arbitrary execution of native code without restriction, leading to a huge proliferation of adware, spyware and other malwares (Barth *et al*). However, modern browsers such as Mozilla Firefox and Google Chrome include security-focused extension platforms.

Carlini *et al* evaluates the Google Chrome extension platform. They immediately note in their abstract that insecure browser extensions "[provide] a way for website and network attackers to gain access to users' private data and credentials". Google Chrome's framework uses privilege separation, jailing and sandboxing via 'isolated worlds', and carefully selected privilege allocation.

Even with these protections, Carlini *et al* exposed vulnerabilities in many extensions and note that more restriction would be valuable. They do, however, also note that extensions must have significant access to be functional - "Extensions can read and manipulate content from websites, make unfettered network requests, and access browser user data like bookmarks and geolocation" - which increases the possible surface of attack.

Sandboxing and privilege separation are also used to good effect in modern solutions that support advertising. AdJail (Ter Louw *et al*, 2010) and AdDroid (Pearce *et al*, 2012) both use sandboxing to isolate users from advertisements, where AdJail is a mechanism for web advertisements and AdDroid is specifically for Android application developers. Both recognise that the advertisement-centric aspects of apps or pages often requires higher privileges (eg. location information) than the content they support; by separating advertising functionality, the privileges for other content can be reduced. Similarly, Google Chrome extensions divide the extension architecture into two components; content script with no privileges, and core extension with full privileges. Content scripts and website are isolated via sandboxing; the website cannot access the program heap of content scripts (Carlini *et al*, 2012).

3. Methods

As part of our analysis of the findings of Provos *et al* we have explored three different practical areas related to privilege escalation. We created a model-slave model implementation in Bash, examined the transformation of code to drop dead privileges, and exploited widely available SUID executables and Linux commands to achieve privilege escalation.

3.1 Monitor-slave model implementation using Bash

We explore the Provos *et al* monitor-slave mode with a pair of sets of shell scripts that implement a simple telnet-style server. One set uses a monolithic design, and the other uses the monitor-slave model. We introduce a vulnerability into both sets of scripts by passing unsanitised user input to eval (a mixing of data and code), then mount an attack on both that is significantly less effective on the monitor-slave implementation due to the inclusion of privilege separation.

We use named pipes for inter-process communication (IPC) and simulate authentication by checking provided usernames against /etc/passwd; while this is not ordinarily a privileged operation, we define it as one in our implementation using chroot.

3.1.1 Monitor

As per Provos *et al* we offload unprivileged operations (including network communication) entirely to slave processes that spawn and manage netcat. This minimises the code running with elevated privileges in our monitor. We also perform basic input sanitisation in the monitor using regular expressions to strip all non-alphanumeric characters from all IPC received from the slave; this prevents simple command injection attacks from a compromised slave and represents improved code audit.

Our monitor implementation performs three privileged operations: authentication of a username using /etc/passwd, the spawning of bash at a privilege level equal to the authenticated user, and a process-termination operation unavailable to the slave. These approximately represent the three categories of request specified by Provos *et al* - "Information", "Change of Identity", and "Capabilities", respectively. We do not experiment with exportation of state.

Our monitor implements the observation made by Provos *et al* that the monitor-slave interface should be rigidly defined. Enforcing a pre-defined sequence of requests requires the monitor to have an internal state; we implement this using a loop that contains a sequence of instructions that can only be executed in the correct order. A slave that makes invalid requests via IPC based on the monitor's current state is immediately terminated by the monitor, reducing the possibility of brute force attacks.

3.1.2 Slave

Our slave process runs as a unique user with minimal permissions. It exists within a chroot that contains only the absolute minimum files needed for the slave to operate. Because the slave does not run as root, it cannot spawn new instances of bash owned by arbitrary users. The chroot jail denies it access to binaries that allow it to kill nc instances it creates, and hides /etc/passwd, which is required for authentication. This mirrors the jailing technique used by OpenSSH, as observed by Provos *et al.*

We have deliberately omitted any input sanitisation within the slave to simulate a potential vulnerability through command injection. However, as the slave is jailed with chroot and communication though the IPC channel is sanitised by the monitor, the compromised slave cannot achieve privilege escalation.

3.1.3 Monolithic implementation

Our monolithic implementation necessarily runs with root privileges so that it can spawn new bash processes as an arbitrary user, which is a privileged operation. It is not contained within a chroot, has full root privileges to modify system configuration, has access to the full suite of system binaries, and has direct access to /etc/passwd, an operation we consider privileged. The monolithic implementation has the same input sanitisation failure vulnerability as our slave script.

3.1.4 Results

The privilege separation technique explored by Provos *et al* was effective at preventing privilege escalation in our implementation. A vulnerability in the user-facing slave did not allow a simulated attacker to execute privileged operations, while the monolithic implementation allowed arbitrary code execution as root and thus access to restricted resources. Further details are available on GitHub.

3.2 Transforming Code to Drop Dead Privileges (Hu *et al*, 2018)

To help programmers write programs that follow the Principle of Least Privilege, operating systems divide the power of the root user into separate privileges which applications can enable on demand. However, using such privileges requires tedious temporal reasoning of program behavior.

3.2.1 Abstract

This section describes a compiler, named AutoPriv, that helps programmers use privileges more easily. AutoPriv uses whole-program analysis during link-time optimization to determine where programs use privileges; it then transforms programs to remove unnecessary privileges during their execution. We tested AutoPriv on several privileged open-source programs and our results show that AutoPriv increases optimization time by 19% on average but exhibits practically no overhead.

3.2.2 Design

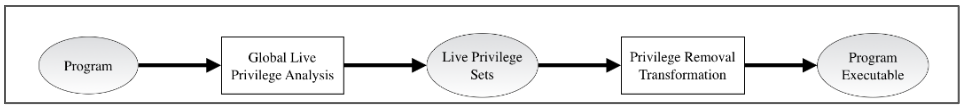

Program
Global Live Privilege Analysis
Live Privilege Sets
Privilege Removal Transformation
Program Executable

Figure 1 AutoPriv Architecture (Source: Hu et al, 2018)

AutoPriv makes a few assumptions about how programs use privileges. First, AutoPriv assumes that each program has a known set of required privileges; privileges are not inherited across programs via the exec() family of system calls. Second, AutoPriv assumes that a programmer has added code to raise and lower privileges. Third, AutoPriv assumes that external library code does not raise and lower privileges.

3.2.3 Privilege Primitives

To provide a simpler interface for manipulating process capability sets, a library is created to wrap the Linux cap_get_flag() and cap_set_flag() system calls. This library provides a set of privilege manipulation primitives that is used throughout the work; ***priv_raise*** temporarily adds saved permitted privileges to the effective set, ***priv_lower*** temporarily removes privileges from the effective set, and ***priv_remove*** removes privileges from the effective and permitted sets permanently.

3.2.4 Implementation

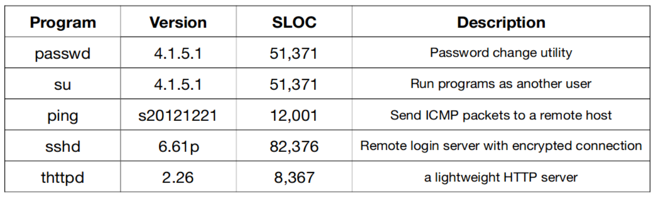
AutoPriv involves the manual addition of *priv_raise()* and *priv_lower()* pairs around privileged system calls and library functions, and 5 LLVM IR passes:

- **Split basic blocks:** Transform the program by splitting system calls bracketed by priv_raise() and priv_lower() calls, as well as calls to internal functions, into separate basic blocks. This simplifies the logic of the local and global privilege analysis passes as well as the privilege removal instrumentation pass.
- **Local Privilege Analysis:** Compute the local privilege analysis results.
- **Inter-procedural Privilege Analysis:** Compute the global privilege analysis results.
- **Inter-procedural Live Privilege Analysis:** Compute the global live privilege analysis results.
- **Privilege Removal Instrumentation:** Add calls to priv_remove.

After running the privilege removal instrumentation pass, AutoPriv uses the existing LLVM *simplifycfg* pass to undo the changes created by the Split Basic Blocks pass. This ensures that AutoPriv does not induce unnecessary overhead due to control-flow graph modifications.

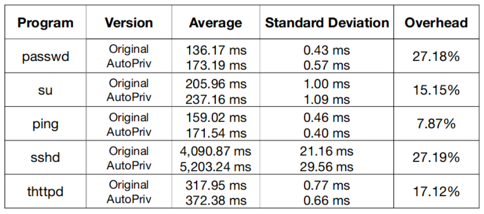
3.2.5 Performance Experiments

To evaluate AutoPriv, we studied both its performance when compiling programs (affected by program analysis and instrumentation) and the performance of programs transformed by AutoPriv (affected by calls raising, lowering and removing privileges). Figures 2 and 3 show the test programs used for evaluation, and compiler overhead for experiments running 20 times on each program; Figure 4 shows the application overhead setup and results.



Program	Version	SLOC	Description
passwd	4.1.5.1	51,371	Password change utility
su	4.1.5.1	51,371	Run programs as another user
ping	s20121221	12,001	Send ICMP packets to a remote host
sshd	6.61p	82,376	Remote login server with encrypted connection
thttpd	2.26	8,367	a lightweight HTTP server

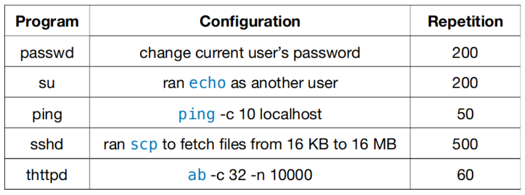
Test Programs



Program	Version	Average	Standard Deviation	Overhead
passwd	Original AutoPriv	136.17 ms 173.19 ms	0.43 ms 0.57 ms	27.18%
su	Original AutoPriv	205.96 ms 237.16 ms	1.00 ms 1.09 ms	15.15%
ping	Original AutoPriv	159.02 ms 171.54 ms	0.46 ms 0.40 ms	7.87%
sshd	Original AutoPriv	4,090.87 ms 5,203.24 ms	21.16 ms 29.56 ms	27.19%
thttpd	Original AutoPriv	317.95 ms 372.38 ms	0.77 ms 0.66 ms	17.12%

Compiler Overhead

Figure 2 (Left) and Figure 3 (Right) (Source: Hu et al, 2018)



Program	Configuration	Repetition
passwd	change current user's password	200
su	ran `echo` as another user	200
ping	`ping` -c 10 localhost	50
sshd	ran `scp` to fetch files from 16 KB to 16 MB	500
thttpd	`ab` -c 32 -n 10000	60

Figure 4 Application Configurations (Source: Hu et al, 2018)

This demonstrates that AutoPriv induces an average overhead of 19% across benchmarks on optimization time and added almost negligible overhead to applications; the difference in bandwidth is within a standard deviation. Furthermore, AutoPriv added almost negligible overhead to applications; the difference in bandwidth is within a standard deviation.

3.3 Privilege Escalation by Exploiting SUID Executables and Linux Commands

Provos et al do not consider other possible means of privilege escalation. Defence-in-depth is important to consider when proposing solutions that prevent these other means. Although privilege separation is an essential primitive to prevent privilege escalation, it is not the ultimate solution, as there are other ways that unprivileged users can still gain a root access. For example, nmap version 3.81 can give a regular user a root shell by simply running nmap in --interactive mode, as it has the SUID bit set to be able to run as root for efficient network scanning (Feroze, 2018). In another recent example, the Linux command sudo has a vulnerability (CVE-2019-14287) in its security policy, the sudoers file. Due to sudoers misconfiguration, normal user could run as a root even if an administrator explicitly restricts the root access from that user. Eventually, reguser can run sudo commands as root by giving a user ID -1 or its unsigned number 4294967295 as [sudo -u#-1 bash OR sudo -u#4294967295 bash] (Miller, 2019). Further details are available on GitHub.

4. Conclusion

Throughout this review we have looked at how privilege escalation occurs and methods for limiting its effects. We have examined several primitives for preventing privilege escalation, all of which fall under the auspices of the principle of least privilege – jailing and privilege separation (as used by Provos et al), sandboxing, and compiling code to automatically drop the dead privileges.

We explored the use of privilege separation in programs that perform privileged operations, and judged the effectiveness of strict, well-defined interfaces for communication between privileged and non-privileged processes. Our own implementation of this model successfully prevented privilege escalation via a deliberately introduced vulnerability, however it reinforced our concerns that the trust placed in the perceived security of a privileged monitor could lead to a failure to observe the principle of defense in depth.

We also commented on the importance of the principle of *defence in depth*, and how improved security gained via these primitives should be stacked with other methods to defend against individual vulnerabilities or weaknesses. Similarly, we expressed our belief that the possibility of privilege escalation of otherwise legitimate users should be evaluated carefully and not discounted as a minor risk.

We further examined how the nature of privilege escalation has changed since Provos *et al* was published in 2003. We explored the proliferation of mobile devices and modular approach to browser extensions, noting how different environments, models of distribution and systems of user interaction affect how we approach privilege-related security in the modern world.

References

Barth, A., Felt, A.P., Saxena, P. and Boodman, A., 2010. Protecting browsers from extension vulnerabilities. In Network and Distributed System Security Symposium (2010).

Brumley, D., & Song, D. (2004, August). Privtrans: Automatically partitioning programs for privilege separation. In USENIX Security Symposium (pp. 57-72).

Carlini, N., Felt, A.P. and Wagner, D., 2012. An evaluation of the google chrome extension security architecture. In *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)* (pp. 97-111).

Chen, H., Wagner, D. A., & Dean, D. (2002, August). Setuid Demystified. In USENIX Security Symposium (pp. 171-190).

Feroze, R. (2018). *A Guide To Linux Privilege Escalation*. [online] payatu. Available at: https://payatu.com/blog_12 [Accessed 12 Nov. 2019].

Guide2Research. 2019. USENIX Security 2019. [ONLINE] Available at: http://www.guide2research.com/conference/usenix-security-2019. [Accessed 13 November 2019].

Hu, X., Zhou, J., Gravani, S. and Criswell, J., 2018, September. Transfo/rming Code to Drop Dead Privileges. In 2018 IEEE Cybersecurity Development (SecDev) (pp. 45-52). IEEE.Wagner, D.A., 1999.

Kilpatrick, D. (2003, June). Privman: A Library for Partitioning Applications. In *USENIX Annual Technical Conference, FREENIX Track* (pp. 273-284).

Mille, T. (2019). *Potential bypass of Runas user restrictions*. [online] sudo. Available at: https://www.sudo.ws/alerts/minus_1_uid.html [Accessed 13 Nov. 2019].

Pearce, P., Felt, A.P., Nunez, G. and Wagner, D., 2012, May. Addroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security* (pp. 71-72). Acm.

Provos, N., Friedl, M. and Honeyman, P., 2003, August. Preventing Privilege Escalation. In USENIX Security Symposium.

Ritchie, Dennis M. "‘*On the Security of UNIX.*" UNIX Supplementary Documents (1979).

Saltzer, J.H., 1974. Protection and the control of information sharing in Multics. *Communications of the ACM*, *17*(7), pp.388-402.

Scholar.google.com. (2019). *Google Scholar*. [online] Available at: https://scholar.google.com/ [Accessed 14 Nov. 2019].

Ter Louw, M., Ganesh, K.T. and Venkatakrishnan, V.N., 2010, August. AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements. In *USENIX Security Symposium* (pp. 371-388).