Hooking Graceful Moments: A Security Analysis of Sudo Session Handling

Ji Hoon Jeong[1,2], Hyung Chan Kim[1(✉)], Il Hwan Park[1], and Bong Nam Noh[2]

[1] The Affiliated Institute of ETRI, P.O. Box 1, Yuseong, Daejeon 34188, Republic of Korea
kimhc@nsr.re.kr

[2] Chonnam National University, Yongbong-ro 77, Gwangju 61186, Republic of Korea

Abstract. *Sudo* is a widely used utility program to temporarily provide the privileges of other users when executing shell commands in many UNIX and Linux systems. In conventional usage, a Sudo user who fulfills password authentication is eligible to execute a series of shell commands with system administrative privilege for a while. As Sudo enables privilege switchover, it has been the attractive target of attacks for privilege escalation in nature. Although Sudo source code have been reviewed by security researchers and patched accordingly, in this paper, we show that Sudo is still vulnerable to session hijacking attacks by which an attacker is able to achieve privilege escalation. We explain how such attacks are possible by spotlighting the inherently flawed session handling of Sudo. We also describe two attack designs – shell proxy and ticket reuse attack – by revisiting some known attack strategies. Our experimental results show that the recent versions of Sudo, in combination with the underlying shell program, are affected to the attack designs.

Keywords: Least privilege principle · Session hijacking · Privilege escalation · Software security

1 Introduction

Sudo [4], firstly developed at around 1980, is a utility program that enables temporal privilege switchover from one user to another in executing shell commands for UNIX and Linux systems. Sudo has been commonly used for administrative tasks, which require system *root* privilege, such as installing/updating software packages, editing system configuration files, adjusting kernel parameters, and so on. Sudo is vastly adopted to be used by default in widely deployed Linux distributions such as Debian, Ubuntu, and its derivatives. The other distributions mostly include Sudo package to be activated by choice. Mac OS X is also configured to use Sudo by default.

The opinions expressed herein reflect those of the authors, and not of the affiliated institute of ETRI.

DOI: 10.1007/978-3-319-44524-3_3

Sudo is usually encouraged as it provides several security benefits. With Sudo deployment, a user does not need to get, and stay with, interactive root shell session to execute a series of administrative commands. Otherwise, the user might input destructive commands inadvertently, thereby impairing the system under management. Moreover, multiple users assigned to administrative jobs do not need to share any root password and their rights to execute a specific set of privileged commands can be separated by configuration. As for accountability, Sudo instance actively logs whole issued commands while conventional root shell session does not.

Despite of the wide adoption, as for attackers, Sudo is the attractive target of attacks to achieve *privilege escalation* in nature. Sudo enables temporal privilege overriding: therefore, it is evident that if an attacker is able to compromise a Sudo user (Sudoer), the attacker has chances to set up an attack to surreptitiously execute shell commands with the additionally permitted privilege. The target privilege is of administrative (root) account in most cases. This kind of attack is feasible with the proxy execution strategy: *i.e.*, the attacker may be able to let the victim user to execute some commands of the attacker's flavor on behalf of the attacker by exploiting Sudo session weakness.

Sudo, in fact, has been considered to prevent this weakness with a ticket-based session management scheme together with some additional protective mechanisms. If a Sudo session is established after taking configured – mostly password based – authentication procedure, then a ticket is issued by which the following Sudo shell command can be executed with no additional authentication until the timestamp expiration in the ticket. Fundamentally, the attacker cannot invoke privileged shell commands directly from another terminal as Sudo checks *session ID* [8] and also validates whether the terminal in use is the place of ticket issue. Sudo also prohibits the well-known *library injection attacks* by throttling some environmental variables which can be abused for replacing a dynamic shared object (dso) with malicious one. This is intended to preclude the possible session hijacking attack by a modified shared object dynamically linked with Sudo. In addition, there are external defense measures provided at kernel layer: *Yama* Linux security module [6] as well as *SELinuxDenyPtrace* [3]. Both are helpful to make difficult or impossible *code injection attack*. Yama restricts the scope of `ptrace` system call and SELinux with ptrace control even turns off the system call facility. These two measures thus can protect shell process tied with valid Sudo session from adversarial code implanting.

Notwithstanding all the defense measures and mitigation efforts, in this paper, we show that it is still possible to achieve successful privilege escalation attack against the very recent versions of Sudo (1.8.16). There have been some previous works concerning the proxy execution and ticket constraint avoidance strategies [13,18,20] to tackle Sudo session handling issues. Basically, here we revisit the two strategies elaborating those to work on the recent Sudo versions. Unlike the previous works, we present more comprehensive and detailed review on session handling issues and also provide concrete attack examples. Note that we do not tackle vulnerabilities of code implementation. Our viewpoint is rather on highlighting the flawed session handling of Sudo.

To show the security impacts of our analysis, first of all, we examine the privilege semantics of Sudo session. The ticket-based session establishment is basically for the convenience: once a Sudo session is established, the validated Sudoer does not need to take repetitive authentication procedures for the following Sudo invocations until the session timeout. One fundamental security issue here is on that the invoking shell process gets to semantically have *dual privileges* which is definitely inappropriate in terms of the *principle of least privilege*.

Although the dually privileged session is inevitable for the purpose of using Sudo, great care must be taken in dealing with established session. The other problem is on that Sudo seems to be failed to prevent session hijacking attacks due to insufficient session protection mechanism. Throughout this work, we found that the following factors can lead to illegal reuse of established – in our views, dually privileged – Sudo session:

– **Lack of integrity check for issuing shell processes.** Sudo does not care whether issuing shell is trustworthy or not. An attacker, thus, is able to execute privileged shell commands by compromising the issuing shell process bypassing the most of Sudo protections.
– **Reusable ticket.** Although Sudo session ticket has been revised to prevent ticket reuse attack, it is still possible to reuse it. If an attacker is successful in matching out ticket parameters within a given session timeout, the ticket is reusable resulting in the illegal session reuse without valid authentication.

It is quite clear that the misuses of established Sudo session may result in escalation from the privilege of Sudo user – and also of the attacker – to that of another – normally administrative – one. Based on the problem review, we describe real world attack designs and conduct experiments on a Linux system installed with the recent versions of widely used distributions.

Contributions. In this paper we will analyze the security impacts of the flawed Sudo session handling and describe realistic attack designs.

1. A study of the improper session handling which may incur dually privileged state in terms of the privilege semantic.
2. Exploring weaknesses around Sudo which may lead to the possible misuse of established Sudo session.
3. Providing two proof-of-concept examples based on attack designs revisiting the shell proxy as well as ticket reuse strategies.

The source code package of our experimental tools, to demonstrate the attack designs, is available on a github project[1].

Paper Organization. Section 2 presents background of Sudo session handling and then, in Sect. 3, we show the problem issues around it. In Sect. 4, we describe attack designs to conduct realistic attacks for privilege escalation and show the experimental results in Sect. 5. In Sect. 6, we discuss on related work. Section 7 concludes this paper.

[1] https://github.com/binoopang/sudo.

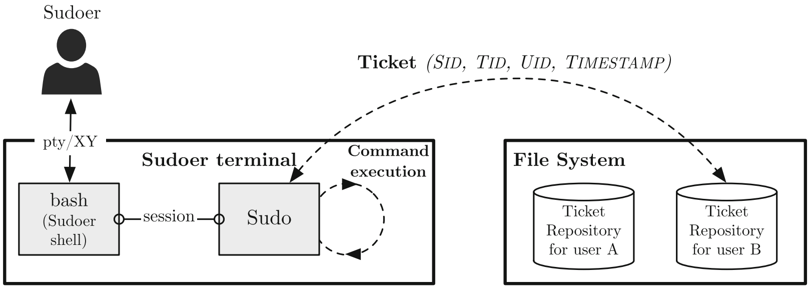

Sudoer
pty/XY
Ticket (SID, TID, UID, TIMESTAMP)
Sudoer terminal
Command execution
bash (Sudoer shell)
session
Sudo
File System
Ticket Repository for user A
Ticket Repository for user B

Fig. 1. The relation among Sudo session entities.

2 Background

2.1 Establishing Sudo Session

Sudo enables a configured user, usually assigned to administrator role, to invoke shell commands with root privilege. In order to invoke a series of privileged commands with Sudo, the user is required to be legitimately authenticated. As Sudo operates on a per-command basis [16], in fact, every command invocation should be passed through proper authentication. At the very first command execution under Sudo, a user is required to undergo, primarily, password based authentication procedure at first hand unless non-default authentication methods are included by configuring *Plugable Authentication Module (PAM)* [10]. Unlike *su* command, which requires the password of root account, Sudo requires the password of Sudoer.

For the sake of convenience, Sudo maintains the ticket-based session management so as not to put Sudoer through repetitive password inputs. After the successful initial authentication, Sudo establishes a session and issues the corresponding session ticket. The following privileged command invocations are legitimately permitted without further password authentication by checking the validity of the issued ticket. The session validity is hold during the *grace period* and it is 5 min in most default configurations.

Throughout this paper, we mean that *Sudoer shell* and *Sudoer terminal* are the respective entities associated with established session. Figure 1 illustrates the relationship among Sudo session entities and it is the conceptual situation after initial authentication.

2.2 Session Tickets

The established, thus valid, Sudo session is associated with a ticket. When a Sudoer passes through initial authentication, the corresponding session ticket is generated and it is stored in the per-user ticket repository. The repository is not allowed to be read and/or modified by non-root users: only root privileged user can access to the repository directly.

The parameters of Sudo ticket has been revised for many years to deal with security issues brought up by public communities. In the latest version of Sudo (version 1.8.16), the two types of tickets are involved: *tty_ticket* and *ppid_ticket*.

tty_ticket. When an user wants to perform a series of shell command executions interactively, the user have to log in into the given system directly via terminal devices such as local console (*tty*), serial (*ttyS*), pseudo terminal (*pty*), and the like. For brevity, here we use the term tty as the representative terminal device instance. When Sudo handles sessions with per-tty granularity, Sudo generates *tty_ticket* for each terminal which has completed initial authentication. The parameters of valid tty_ticket is comprised of quadruple (S_{ID}, T_{ID}, U_{ID}, $T_{IMESTAMP}$); The user U_{ID} associated with the session S_{ID} and the terminal T_{ID} is permitted to invoke privileged shell commands. The tty_ticket is valid during the *grace period*, *i.e.*, from the time of ticket issuing $T_{IMESTAMP}$ to the time $T_{IMESTAMP} + timeout$ where the *timeout* is 5 min in default configuration. S_{ID} is the session ID which is determined by the process ID of session leader. Note that shell process, such as `bash`, becomes session leader in usual Sudo usages. T_{ID} represents the actual device number of (pseudo) tty by which one can discriminate each terminal device in use. U_{ID} is the system user ID of ticket owner (Sudoer). The timestamp field $T_{IMESTAMP}$ records the time of the most recent shell command execution with Sudo. For every Sudo invocations, the $T_{IMESTAMP}$ value is updated accordingly.

ppid_ticket. It is also possible to deploy Sudo with non-interactive manner. Terminal devices therefore are not directly involved for this case. For example, a user can set up a script that includes login facility with the help of *sshpass* [2]. In such case, Sudo employs the P_{PID} (parent process ID, or ppid) instead of the T_{ID} to establish sessions with per-ppid granularity. The parameters of ppid_ticket are defined by (S_{ID}, P_{PID}, U_{ID}, $T_{IMESTAMP}$); The user U_{ID} associated with the session S_{ID} and the parent process P_{PID} is permitted to invoke privileged shell commands. As same with tty_ticket, ppid_ticket expires session if no further Sudo invocation follows within a given grace period.

3 Problems of Sudo Session Handling

We have conducted an empirical source code auditing as well as dynamic analysis on Sudo program. In this section, we explain our examination results around Sudo session handling issues.

3.1 Dually Privileged Session

The most good reason of Sudo deployment in terms of security is that Sudoer is relatively safe from inadvertent mistake in executing shell commands mixed with root and non-root privileges. This is evident compared to the case of just staying with root shell. Sudoer should be explicit when using the root privileged commands. In view of Sudoer, it looks like explicit privilege switchover between

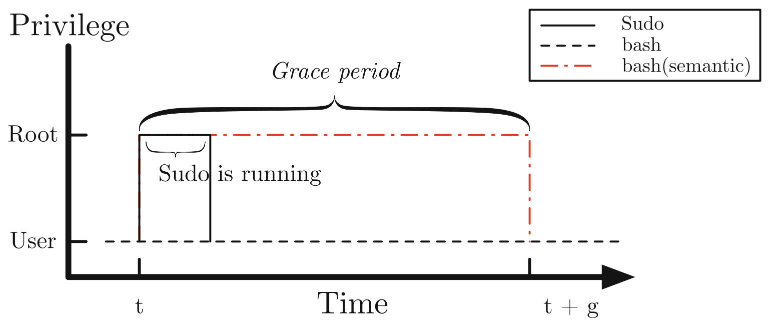


Fig. 2. The privilege transition in a Sudo session.

root and non-root modes. However, this is not true in terms of the privilege semantic. To clear out the privilege transitional semantic in a Sudo session, we illustrate the situation in Fig. 2. Here we define a function $max\_priv(a, b)$ which returns the maximum privilege between the time a and b. Then, $max\_priv(t, t+g)$, where t is the time of successful initial authentication (*i.e.*, session start time) and g is the configured grace time (5 min in default), will constantly returns the maximum privilege between the two time points. This implies that the established Sudo session is associated with *dual privileges*: the one of Sudoer and the other one of root account.

The session semantically tied with dual privileges is obviously problematic in terms of the classical *least privilege principle*. In other words, during the given grace period, although the Sudoer shell process is non-root privileged as for what the underlying OS kernel understands, it is actually dually privileged; therefore, the attacker may be able to misuse root privileged commands if one can properly set up the link point between the privileges.

As for the real world link point, the *code injection* technique using `ptrace()` system call could be a candidate in realizing Sudo session attack. More concretely, malicious code injection into Sudoer shell process will incur indirect root-privileged command executions by an attacker who set up the link point between the privileges. In Sect. 4.2, we will present an example scenario of this case.

3.2 Too Wide Subject Eligibility

When Sudo is invoked, in normal cases, there would be a shell process, such as *bash*, as its parent. The shell process might be initially created by `fork()` call from, *e.g.*, login, sshd, or gnome-terminal process or it can be created from another shell process, especially if Sudo is invoked from a script file. If a given shell process creates some child processes, these are tied with the same session ID (sid) [11][2]. Unless the process does not declare to be a new session leader, it will be included to the same session group with the invoking shell (parent process).

[2] Note that this is not the concept of Sudo session but that of process session.

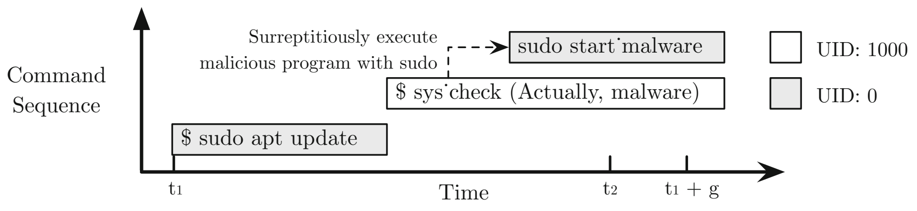

Command Sequence
Surreptitiously execute malicious program with sudo
sudo start_malware
$ sys_check (Actually, malware)
$ sudo apt update
UID: 1000
UID: 0
t1
Time
t2
t1 + g

Fig. 3. Subject eligibility problem: the downloaded malware is able to execute Sudo shell command without authentication after one of the other child of parent shell process establishes a valid Sudo session.

Concerning with the two session concepts, there is the *subject problem*: if one of process once establishes a Sudo session, all the other processes in the same process session group are eligible to invoke Sudo commands without requiring any independent authentication during the given grace period. Evidently, it is not safe because an arbitrary child process with the same process session ID can invoke root privileged commands and it possibly leads to security problem if the child is compromised.

Let's take an example of such negative case, shown in Fig. 3, assuming that a user downloaded a system utility of one's flavor. Opening a terminal, the user makes system changes using some administrative commands (`apt`) with Sudo. Then, in the same terminal, the downloaded utility is invoked (`sys_check`) to evaluate newly changed environment *without Sudo* as it does not require root privilege. At this time, the utility, which is actually malware implanted by an attacker, can gain root privilege if it surreptitiously executes malicious Sudo commands because it has the same process session ID with the invoking shell. This situation is surely in contrast to the user's intention, in terms of privilege assignment, because the user executes the utility explicitly without involving Sudo. However, the malicious code in the utility is able to implicitly trigger the malicious commands with root privilege.

3.3 Reusable Ticket

If a session ticket is issued after authentication, it is stored in the local file-system until Sudo reissues the ticket or the system faces to shutdown. Meanwhile, the Sudo session can outlive the shell login session of Sudoer [5] and this means that the Sudo session information is still remained for a while as the ticket is accessible by the file system operations. As a result, an attacker can ride on the outlived Sudo session if the attacker is able to reuse the stored ticket.

Reusing Sudo ticket is trying to match up the shell environment of attacker to the parameters of Sudo session ticket currently saved in the file system: in fact, this is a well-known security issue. However, no complete fixes are yet applied to mitigate this issue as it is generally considered that reusing Sudo ticket is difficult to achieve. In this work, we highlight that Sudo session hijacking attack through ticket reuse trials can be practically achievable.

One ideal case of reusing ticket may happen when Sudoer exits one's shell session immediately after invoking Sudo legitimately, and such usage pattern is commonly occurred in general. For example, a single one-liner Sudo command can be used to manage multiple remote servers with `ssh` command (*e.g.*, `ssh-t admin@host "sudo/path/to/script"`). In this case, an attacker can try to reuse the generated Sudo ticket during the nearly full grace period. In targeting tty_ticket type, defined in Sect. 2.2, the attacker have to match up the two constraints, S_{ID} and T_{ID}, as the user ID of shell process of the attacker is same with U_{ID} by our assumption. The match up should be finished before the time constrained by $T_{IMESTAMP}$.

The attacker can be more aggressive if a Sudoer does not exit the login shell in use by which an attacker cannot resolve `tty` constraint T_{ID} as the Sudoer holds the `tty` by which the underlying OS cannot yield the `tty` value to the attacker. In such case, the attacker can forcibly eject the Sudoer shell process – thus the `tty` is released – and have a chance to resolve the T_{ID}. It would be more effective if the ejection is performed right after the successful Sudo invocation, as the attacker can have sufficient grace period to match up the related parameters for reusing the Sudo session ticket.

4 Attack Designs

This section presents the possible attack designs based on our examination. We basically revisit strategies discussed in some previous works as well as related communities around Sudo. Here we limit our environmental boundary to Linux systems and confine to the privilege escalation case – *i.e.*, from non-root to root – for the sake of succinctness.

4.1 Attacker Model and Assumptions

We assume that an attacker is successful in breaking into a victim system and gets the privilege of Sudoer via any possible scenarios except password sniffing. The scenarios may include performing APT style attacks such as sending malicious e-mail and/or performing watering hole scenarios [12,21]. By the first stage attack, the attacker is successful in driving to download an infected upgrade package, or any other methods to gain privilege of the victim account, so as to execute shell commands without the knowledge of the password of that account. The attacker's goal here is achieving privilege escalation from the victim account to the system root account. Real world attack campaigns may also have the similar goal.

Sudo attack designs require such a strong assumption as it is not a server-like remotely exploitable program. Rather, privilege escalation exploiting Sudo might be the second stage goal for attackers within a local system. Moreover, under this strong assumption, there would be alternative ways to get the root privilege excluding Sudo involvement such as inducing to execute a fake version

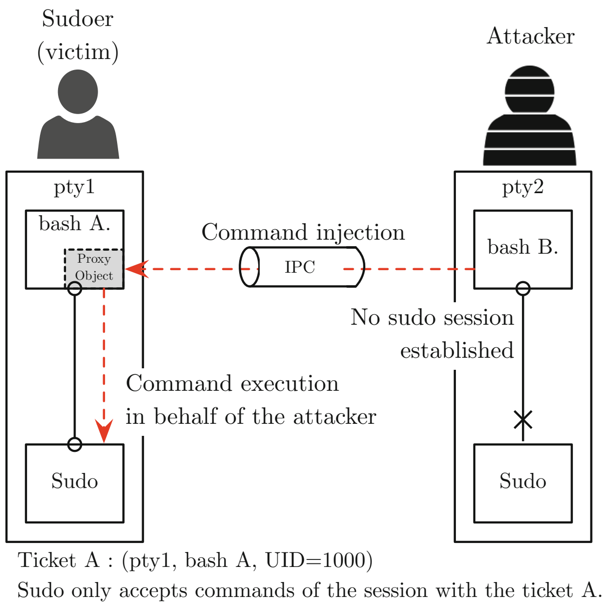

Sudoer (victim)
Attacker
pty1
bash A.
Proxy Object
Sudo
Command injection
IPC
pty2
bash B.
No sudo session established
Command execution in behalf of the attacker
Sudo
Ticket A : (pty1, bash A, UID=1000)
Sudo only accepts commands of the session with the ticket A.

Fig. 4. Shell proxy attack.

of Sudo. However, our aim is here to review the Sudo session handling issues, therefore, the attack models of our interests inevitably involve Sudo.

Specifically, if the attacker successfully access the target victim shell with any first stage attacks, the victim account should meet the following conditions:

- The account have read and write permissions in accessing local file system and this is normally true in most cases.
- The account is included in the Sudoer group. This means that the account is eligible to execute privileged shell commands with Sudo.

4.2 Shell Proxy Attack

Here we present a proxy style attack to invoke arbitrary shell commands with root privilege. Figure 4 depicts the flow of shell proxy attack. The core idea behind this attack is on compromising live Sudoer shell process so that make the Sudoer shell process execute some commands of the attacker's flavor. The attacker may set up the *shell monitor* that detects successful Sudo command invocations. The setup also includes the *proxy executor* that literally executes attacker's Sudo command. Whit this setup, the attacker's commands can be executed with root privilege bypassing tty_ticket constraint.

In order to achieve shell proxy attack, the attacker must be able to inject proxy code into the address space of the Sudoer shell process. This is basically permitted as the shell processes of the attacker and Sudoer have same user ID. There are a few techniques to perform code injection: the attacker can make use of the debugger purposed system call (*e.g.*, `ptrace()`) or the linker trick taking advantage of the environmental variable (`LD_PRELOAD`). The former enables the attacker to inject proxy code directly and the latter might be used for the shared object, which includes proxy code, to be loaded into the shell process of the victim. In our test case, we first build a proxy shared object to deliver the attacker's commands as well as to execute them in behalf of the attacker. The channel between the proxy object and the attacker can be set up with inter-process communication (IPC) facility such as *pipe* [7].

The followings further explain how the attacker can inject the shared object, which contains proxy functions, into Sudoer shell process.

Shared Object Injection with Ptrace System Call. The process tracing system call (`ptrace()`) is like a swiss army knife for many hackers as it provides rich interface to control over live processes considerably. With the `ptrace()`, one can pause a process, change register values, and even patch data and/or code part of the process. The daily use of debugging tools such as *gdb* and *strace* are implemented on top of the `ptrace()` system call. Meanwhile, attackers are also in favor of this facility to patch and/or inject malicious code/object.

As for injecting shared objects into a live process, there is a famous system API function in Windows world: `CreateRemoteThread()`. Attackers can use this function perform shared object injection very easily. On the other hand, in Linux/Unix world, there is no such system call or function supported by underlying OS or system libraries. However, there are some efforts to enable shared object injection such as Jugaad [9] and hotpatch [14]. These works also internally use `ptrace()` to inject shell code which eventually loads shared object into target process.

Shared Object Injection with Linker Trick. In Linux systems, there is a well known linker trick by which one can inject shared object into initiating process. The trick is supported by the linker (*ld*). It can be used with the environmental variable *LD_PRELOAD* that points to the path of shared object which is loaded preferentially. This is widely used to extend features or to modify behavior of pre-built application without re-compiling main executable.

In order to implant the proxy object into Sudoer shell, we modify the initialization script of the shell, `.bashrc` in case of the shell program *bash*. Whenever bash is started, by Sudoer login, `.bashrc` script in the Sudoer's home directory is read and the according initialization is applied for the bash process. Note that most other shell program such as *csh* and *ksh* also deploy such initialization script. By the attacker model of this work, the attacker can modify `.bashrc` file as the shell session of the attacker is associated with the same user ID (uid) with that of victim Sudoer.

```
if [ -z "$SUDO_EXP" ]; then
    echo "First execution of bash"
    export LD_PRELOAD=/tmp/libshproxy.so
    export SUDO_EXP=TRUE
    bash
else
    echo "Second execution of bash"
    unset LD_PRELOAD
fi
```

Listing 1.1. Code snippet that loads our proxy object automatically when a bash process is started.

In our attack setup, the code snippet in Listing 1.1, is inserted to `.bashrc` file of the victim Sudoer. After that, as soon as the Sudoer logs in into the system, two bash processes will be started in series: the first one will execute the second bash by our code resulting in the injection of our proxy object specified with LD_PRELOAD (*i.e.*, /tmp/libshproxy.so).

4.3 Ticket Reuse Attack

As explained in Sect. 3.3, Sudo session ticket is able to be reused, and therefore, Sudo session is also reusable under if successfully resolved. Let's assume that an attacker is targeting a just established Sudo session of the tty_ticket (S_{ID},U_{ID},T_{ID},$T_{IMESTAMP}$), and then the Sudoer is logged out by oneself or kicked out by the attacker.

In order to reuse the Sudo session, the attacker needs to resolve the constraints of each parameters of the ticket before the ticket expiration. As the attacker has valid U_{ID} by the attacker model, the S_{ID} and T_{ID} should be resolved to be successful in session reusing. Basically, the attacker can resolve the two constraints by matching up the attacker's shell environment with a simple brute force manner. The attacker repeats terminal process creation until the values of the two parameters are matched up. During the match up procedure, the S_{ID} and T_{ID} can be compared with the values obtained from results of `ps` command invocation.

Resolving T_{ID}. T_{ID} is a minor number of pseudo terminal device and all pseudo terminals have the unique T_{ID} for them. That is why the T_{ID} is used as a index parameter for getting associated tty_ticket. In fact, Sudo stores both major numbers and minor numbers of pseudo terminal devices in the ticket storage. It is not necessary to match up major number as all those values are same. However, each pseudo terminal has unique minor number.

Since Linux system tries to assign the lowest value to T_{ID} like `open()` system call's fd assignment, we can predict the next T_{ID} value for the newly created pseudo terminal. Therefore, an attacker is able to match up the T_{ID} constraint by simply creating a new pseudo terminal after the shell process of victim Sudoer is just exited (or forced to be exited). This approach guarantees that the attacker will eventually get the same T_{ID} value for one of newly created terminals because all the non-assigned values can be tried by the attacker.

Resolving S_{ID}. S_{ID} is a session ID determined by the process ID of session leader. Sudo compares the current S_{ID} with the one in a given ticket and reject any Sudo invocation if the S_{ID} does not match to prevent the class of session hijacking attacks as well as session reuse attacks [18]. As the probability of having the same session ID is $\frac{1}{32767}$ in typical 32-bit systems, it is hard to reuse sessions with manual Sudo tactics. However, a simple brute force approach – which is similar with T_{ID} resolving – will also work for resolving the S_{ID}.

The domain of process ID is a finite totally ordered set $< P, \leq >$ where P is defined by $\{x \mid (1 \leq x) \wedge (x \leq 32767)\}$ in 32-bit Linux systems. The process ID assigning function f returns from the least element 1 and continues to increase, by consecutive calls, until it reaches to the greatest element 32767. If the return of f reaches to that value, then the next return value will roll over. Thus, the attacker is able to iterate the whole process ID domain except the ID values already assigned and the target ID value for S_{ID} will be matched up eventually.

A Brute Force Algorithm to Revive Sudo Session. Attackers can think of the process ID assigning function f to be `fork()` system call in Linux systems. `fork()` creates a child process and a new process ID will be assigned to the child. In order to resolve S_{ID} and T_{ID} constraints at the same time, attackers can deploy `forkpty()` which is a combination of `openpty()`, `fork()`, and `login_tty()`. Thus, it has the ability to generate the values for process ID as well as terminal ID. With the system call, attackers can try to perform the match up process for both S_{ID} and T_{ID}.

With the `forkpty()` system call, we devise a simple brute force algorithm (Algorithm 1) to resolve S_{ID} and T_{ID} constraints. At the first phase, T_{ID} resolving is performed with a simple brute force loop. If the target T_{ID} value is obtained, it is forced to close the terminal immediately so that the same T_{ID} value can be produced again in the very next `forkpty()` call. At the 2nd phase, `forkpty()` is called repeatedly to get the targeting value for S_{ID}. When it returns the right S_{ID}, the shell with the revived Sudo session will be ready. Note that this algorithm will not fail unless one of the other processes coincidentally pre-occupies the targeting values by assigning a new terminal or forking a new process in the middle of the loops.

5 Experimental Evaluation

This section reports our evaluation results of the two attack designs – shell proxy and ticket reuse attack – described in the previous section. To confirm the security impacts of the attacks, we implemented a set of tools to realize the two attacks and tested on a dozen of Linux systems. The Linux distributions were selected by the popularity referencing the distrowatch web site [1].

5.1 Experiment I: Shell Proxy Attack

Shell proxy attack (Fig. 4) can be performed with the linker trick explained in Sect. 4.2. One can build a shell proxy object, which is a basically `dso` with

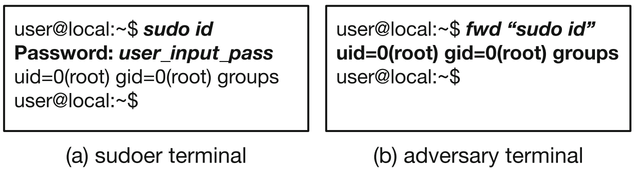
Algorithm 1. Resolving S_{ID} and T_{ID} constraints.

```
$S_{ID}$: A target session ID.
$T_{ID}$: A target pseudo terminal ID.
fd  : A file descriptor to communicate with the terminal of $T_{ID}$.

▷ 1st phase: try to resolve the $T_{ID}$ first.
while true do
    tid ← CreatePseudoTerminal();
    if tid equals to $T_{ID}$ then
        ClosePseudoTerminal (tid);
        break ;

▷ 2nd phase: here we have the valid $T_{ID}$, now resolve the $S_{ID}$.
while true do
    (sid, fd) ← forkpty();
    if sid > 0 then
        if sid equals to $S_{ID}$ then
            return fd ;
        else
            close(fd);
    else
        ▷ If the two constraints are matched up, child process invokes execve()
        to make the shell environment for reusing Sudo ticket.
        if sid equals to $S_{ID}$ then
            execve(bash);
```

position independent code (PIC) option enabled. When the proxy object is loaded into a shell process of interest, it first creates a thread to perform proxy activities while the host (shell) process is alive. The role of our proxy function is simple: it just monitors a pipe which is created to receive attacker's shell commands. If a command is pended to the pipe, the monitor just executes it by forking a child process. We also made a tiny commander, named as `fwd`, which actually pushes the monitored shell commands into the communication pipe.

On completion of the attack setup, now we can launch shell proxy attack just by waiting victim Sudoers. Once a Sudoer logs in into the system under test, the proxy object is loaded into the Sudoer's shell process. Now if the Sudoer executes a root shell command "`sudo id`" passing through successful password authentication (Fig. 5(a)), the attacker is also able to execute the same command in the different terminal without password authentication (Fig. 5(b)). To be practical, the very first forwarded shell command might be the one for being a root shell session so as to get the whole control of the system without requiring no more shell proxy attack stages.

Te investigate the possible attack deployment range, we have conducted the same experiment described above on some popular Linux distributions (Table 1).



```
user@local:~$ sudo id
Password: user_input_pass
uid=0(root) gid=0(root) groups
user@local:~$
```

(a) sudoer terminal

```
user@local:~$ fwd "sudo id"
uid=0(root) gid=0(root) groups
user@local:~$
```

(b) adversary terminal

Fig. 5. After the victim user's sudo execution in terminal (a), the attacker placed in terminal (b) is also successful in invoking `sudo ID` command indirectly via `fwd`.

Table 1. The results of shell proxy attack (sorted by the distrowatch hit rank).

Distribution	Default Sudo version	Process protection	Vulnerable?
MintOS Cinnamon	1.8.9p5	ptrace_scope	yes
Ubuntu 15.10(stable)	1.8.15	ptrace_scope	yes
Ubuntu 15.04(stable)	1.8.9p5	ptrace_scope	yes
Debian 7(stable)	1.8.5p2	none	yes
Debian 8(testing)	1.8.10p3	none	yes
Debian 8(unstable)	1.8.11p2	none	yes
OpenSUSE 13.2	1.8.10p3	none	yes
Fedora Core 21(stable)	1.8.8	none	yes
Fedora Core 22(unstable)	1.8.12	ptrace_scope	yes
CentOS 7.0	1.8.6p7	none	yes
RHEL 7.1	1.8.6p7	none	yes

We could be successful in privilege escalation with shell proxy attack for all the distributions we have tested.

5.2 Experiment II: Ticket Reuse Attack

Based on the attack design described in Sect. 4.3, we have performed *ticket reuse attack* on the various versions of Sudo. Here we show our experimental results to answer the two questions: (1) whether it is possible to resolve constraints in Sudo session ticket, and (2) if so, whether attacker is able to finish resolving constraints assigned to session ticket within limited grace period.

Resolving Ticket Constraints. In order to reuse a session ticket, an attacker should be able to resolve all the constraints of the ticket during the grace period defined with $T_{IMESTAMP}$. It would be sufficient time if Sudoer exists or is kicked out from one's shell session just after invoking a valid Sudo command. Otherwise, it depends on the remaining time.

To perform constraint resolving, we built a simple brute force program reflecting Algorithm 1 and applied the program for various Sudo versions compiled from

Table 2. Our experimental results for resolving Sudo ticket constraint (S: Session ID, T: Terminal ID, U: User ID, TS: Timestamp, G: Group ID, CT: Creation Time of PTY).

Version	Release date	tty_ticket	Resolved?
1.8.16	2016-03-17	(S, T, U, TS)	yes
1.8.15	2015-11-01	(S, T, U, TS)	yes
1.8.12	2015-02-09	(S, T, U, TS)	yes
1.8.10	2014-03-10	(S, T, U, TS)	yes
1.8.9	2014-01-06	(S, T, U, G, TS)	yes
1.8.6	2012-09-04	(S, T, CT, TS)	partial

respective official source archives. Table 2 shows our results. We could resolve ticket constraints completely with the version 1.8.7 and above. Meanwhile we were not fully successful with the version 1.8.6 and below. Especially matching up the CT constraint, which is the creation time of pseudo terminal, was not made. In order to match up the CT constraint, attackers should be able to control system local time, however, it deviates from our assumptions. Overall, ticket reuse attack is valid for the most recent versions of Linux distributions as they mostly deploy the vulnerable versions. Note that CentOS and RHEL systems with the major version 7 are not affected as those distributions still stay with Sudo version 1.8.6. However, at the time of this writing, we conjecture that it will be upgraded to some of the vulnerable versions.

As we mentioned in Sect. 2, the parameter constitution of Sudo ticket has been revised for a long time. The current constitution is concretized after the version 1.8.10 and this version newly introduces the *ppid_ticket*. This paper only deal with the tty_ticket case although we also have finished our tests for ppid_ticket case. The results were quite similar in many aspects with the tty_ticket case, thus, we do not include the ppid_ticket results due to the paper limit.

The Feasibility of Reusing Session Ticket. Even though one can resolve a given ticket constraints, it is useless if it takes too much time exceeding the limited grace period. It takes more time to resolve S_{ID} than T_{ID} since the domain of S_{ID} is greater than that of T_{ID}. Furthermore, attackers need to iterate the whole domain of S_{ID} in the worst case. The probabilities of successful reusing both tty_ticket and ppid_ticket are highly dependent on the size of S_{ID} domain and the remaining time until session expiration.

We have analyzed the size of S_{ID} domain for 32-bit and 64-bit Linux systems and it turns out that it was 32768 for 32-bit and 131072 for 64-bit systems respectively. As having larger domain size, attackers need more time to resolve ticket constraints in 64-bit systems.

To measure the constraint resolving time, we have performed our experiments on a virtual machine which has the specification of 2.7 GHz Intel Core i7 and

Table 3. Sudo vulnerabilities reported in the last 5 years

ID	Score	Details
CVE-2014-0106	6.6	Improper environment variable checking in env_reset
CVE-2013-2777	4.4	Improper controlling terminal validation
CVE-2013-2776	4.4	Improper controlling terminal validation
CVE-2013-1776	4.4	Improper controlling terminal validation
CVE-2013-1775	6.9	Bypass intended time restriction by setting the system clock
CVE-2012-3440	5.6	Allows overwriting arbitrary files via a symlink attack
CVE-2012-2337	7.2	Improper support of configurations that use a netmaks syntax
CVE-2012-0809	7.2	Format String Bug. Allows local user to execute arbitrary code
Bug #87023	-	Bypass *tty_ticket* by revisiting closed pseudo terminal

2 GB RAM. Our experiment results show that it takes no more than 20 s in a 32-bit system. With a 64-bit system, we could iterate whole S_{ID} domain within 80 s in our implementation. This means that we can resolve ticket constraints if the remaining time of grace period is greater than, roughly, 2 min in worst case.

However, attackers can set up an attack environment so as not to run the 1st phase of Algorithm 1 by forcibly kicking out Sudoer as soon as the valid Sudo session is detected. With such tactics, the brute force can be more feasible. Unfortunately the 2nd phase loop is hard to be eliminated due to frequent background process launchings and the loop iteration is required in most cases.

In Linux systems, the size of S_{ID} domain can be tuned as it is a kernel parameter (`/proc/sys/kernel/pid_max`). This kernel parameter can influence on the capability of Algorithm 1. In our test, if we set the kernel parameter greater than 451,000, then the constraint resolving iteration could not be completed until session expiration. However, note that the default value of the parameter rarely changed and most Linux distributions retain original value encoded in Linux kernel source code.

6 Related Work

6.1 Sudo Vulnerabilities and Fixes

Since the year 1999, several Sudo vulnerabilities have been reported and we list the related bug report items of the recent years in Table 3.

Bug #87023 [13] discussed the possibility of Sudo session reuse on tty_ticket with reusable T_{ID} which is of previously closed pseudo terminal. As the investigated Sudo version (1.6.8) checks only T_{ID} and U_{ID}, it is possible to reuse tty_ticket if one can reopen a terminal which is associated with the previously used T_{ID} before session expiration. This bug has been dealt with by introducing a new constraint, *i.e.*, the creation time of pseudo terminal (CT). We think that this fix trial can not effectively counter ticket reuse attack as the creation time of real tty is always same after system boot up. Furthermore, attackers may roll back the system time by exploiting *ntp* vulnerabilities [15] to avoid the constraint. Interestingly, the CT constraint is omitted from the version 1.8.9.

CVE-2012-0809, CVE-2012-2337, and CVE-2012-3440 are caused due to improper code implementation. In such cases, the respective code snippets were patched. CVE-2013-1775 is about bypassing ticket constraint in which clock resetting technique is introduced [17]: when a user issues `sudo-k` to expire one's ticket, the time-stamp is set to the UNIX epoch time value. If an attacker can reset the time-stamp to that value, then the attacker also can run Sudo command with escalated privilege without authentication. This vulnerability has been fixed in a way of ignoring timestamp of the epoch value instance.

Among the assigned items, CVE-2013-1776, CVE-2013-2776, and CVE-2013-2777 were related to session handling problems. These are basically the same vulnerability; namely, these have been discriminated because of differently affected versions. These items are related to ticket constraint avoidance issue in where terminal device ID spoofing technique is involved. This issue also has been fixed: session ID is included into the ticket data structure. Since attacker can not spoof his or her session ID, the applied fix is on right way to defeat the some ID spoofing attacks. However, it is possible to reuse ticket by restoring terminal ID and session ID at the same time conducting brute-force attack and, in the last section, we showed the attack setup is realistic through our experiments.

Sudo have been revised to deal with session hijacking problem. Before the version 1.7.3, Sudo does not enable *tty_ticket* by default and which means that one ticket is valid for all of terminal instances. An attacker, thus, could gain root privilege when an user establishes Sudo session. To address this problem, Sudo enables *tty_ticket* enforcing by default after that.

Another notable fix against ticket reuse attack is also made. Before the version 1.8.6, ticket does not includes S_{ID} to classify controlling terminal devices. Since the lack of S_{ID}, an attacker is able to hijack the Sudo session if the attacker can spoof any valid T_{ID}. A related work [18] described the possible T_{ID} spoofing issue in which techniques of redirecting standard file descriptors were presented. To deal with this problem, after the version 1.8.6, Sudo ticket contains S_{ID} and both T_{ID} and S_{ID} should be matched.

6.2 Works on Sudo Session Handling Issues

Sudo Session Hijacking. Napier discussed the issues of least privilege in using some automated tools including Sudo [20]. Especially, he pointed out the improper Sudo session handling: a single Sudo ticket can be used for multiple terminal instances and a established Sudo session can be reusable after the associated Sudoer has logged out. Before version of 1.7.3, Sudo does not enables tty_ticket policy by default. An user, thus, could invokes privileged command over all instance of terminal after one valid session is established. Before version of 1.8.6, tty_ticket does not contain session ID. An attacker, thus, could reuse ticket by login same terminal.

Authentication Bypass Using Terminal ID Spoofing. Castellucci presented that the use of `ttyname()` could lead to the potential bypass of the *tty_tickets constraints* [18]. Sudo deploys `ttyname()` function to discriminate among terminal devices. However, the function accepts file descriptors as inputs for its functionality. If an attacker is able to manipulate the mappings between file descriptors and active terminal devices, he or she can bypass the constraints on tty_ticket. This vulnerability is assigned to CVE-2013-1776.

Environmental Variable Injection. Macke reported that the `env_reset` option is disabled in Sudoers file of Sudo versions until 1.8.5 [19]. When the option is disabled, it is not possible to block out all dangerous environment variables. By this weakness, an attacker could deploy the notorious LD_PRELOAD environmental variable to execute arbitrary commands with escalated privilege. This vulnerability is assigned to CVE-2014-0106. As a workaround, the recent version of Sudo deploy the *proc file system* as well as `sysctl()` instead of relying on `ttyname()`.

Concerning the well-known weaknesses that we discussed above, the latest version of Sudo (1.8.16) activates tty_ticket, by default, to prevent the execution of Sudo from different terminal devices. Moreover, the data structure of session ticket has been revised to contain POSIX session ID to prohibit the known ticket reuse attack. Although such mitigation efforts have been discussed and patched in the related communities, we have shown that it is still possible to override Sudo session by elaborating the attack strategies.

6.3 Possible Mitigations

Disabling the concept of grace period in Sudo session may require consecutive authentication activities for every command invocations and that will lead to usability sacrifice. To sustain the current usage of Sudo, privilege escalation with the attack designs shown in this paper should be mitigated in some ways.

To protect Sudo from a class of shell proxy attacks, we need to disable code injection facilities, such as `ptrace`, more radically with the help of kernel layer component: *e.g.*, applying `ptrace_scope` to any launching Sudo instances. It may be difficult to protect code injection solely with user level solutions. Moreover, it is necessary to supplement invalidating procedure of the used tickets so as to properly counter ticket reuse attacks. For example, introducing some unrecoverable variables, such as nonce values or session creation time, into session ticket items may severely hinder attacker's reconstruction of valid ticket constraints.

Overall, it is certain that Sudo itself does not have whole responsibility to protect attacks for privilege escalation. Because it is quite difficult to protect Sudo itself especially against attacks involving code injection. Moreover, privilege information stored in places accessible by user level process may be susceptible to be recycled and that leads to possible session reuse. Fundamentally, it would be more appropriate to provide a facility for transient privilege switchover as a OS service.

7 Conclusion

In this paper, we presented our security analysis of the vastly used utility program, Sudo, for daily uses to perform some administrative tasks in UNIX/Linux systems. We have conducted empirical source code auditing as well as runtime testing, and noticed that there are still session handling problems in Sudo: dually privileged session, process subject eligibility, and reusable ticket problems. Based on the recognition of these problems, we described the two attack designs to demonstrate effective attacks to hijack live Sudo session so as to achieve privilege escalation. The attack strategies are basically based on previous studies, and we have revisited to make concrete example cases. We have also conducted experiments and confirmed that the attack designs are feasibly deployable by imaginary attackers if they can access to a victim Sudoer shell process but without knowing authentication password.

Possible future work may include enhancing the security of Sudo session handling scheme to mitigate attacks demonstrated through this work possibly reflecting our remarks in Sect. 6.3. Moreover, another approach would involve designing a new scheme to support transient privilege escalation in more safe way replacing the current session handling scheme, and the effort should sustain both usability as well as security.

Acknowledgments. We would like to thank Kaan Onarlioglu, Erwan Le Malécot, and the anonymous reviewers for their suggestions and comments.

References

1. Distrowatch page hit ranking. http://distrowatch.com/dwres.php?resource=popularity
2. Non-interactive SSH password auth. http://sourceforge.net/projects/sshpass/
3. Selinuxdenyptrace (fedora features). https://fedoraproject.org/wiki/Features/SELinuxDenyPtrace
4. Sudo main page. http://www.sudo.ws/
5. Sudoers manual. http://www.sudo.ws/sudoers.man.html
6. Yama, limux security module. http://www.kernel.org/doc/Documentation/security/Yama.txt
7. pipe(7) linux user's manual (2005)
8. credentials(7) linux user's manual (2008)
9. Jugaad: Linux Thread Injection Kit. Defcon19 (2011)
10. Morgan, A.G., Kukuk, T.: The linux-pam system administrator's guide Ver. 1.1.2 (2010)
11. Kerrisk, M.: The Linux Programming Interface. No Strach Press, San Francisco (2010)
12. Kindlund, D.: Holyday watering hole attack proves difficult to detect and defend against. ISSA J. **11**, 10–12 (2013)
13. kko: sudo option "tty_tickets" gives false sense of security due to reused pts numbers (2007). https://bugs.launchpad.net/ubuntu/+source/sudo/+bug/87023
14. Kumar, V.N.: Hotpatch (2013). http://selectiveintellect.com/hotpatch.html

15. Malhotra, A., Cohen, I.E., Brakke, E., Goldberg, S.: Attacking the network time-protocol. IACR Cryptology ePrint Archive 2015, p. 1020 (2015). http://dblp.uni-trier.de/db/journals/iacr/iacr2015.html#MalhotraCBG15
16. Miller, T.C.: Sudo in a nutshell. http://www.sudo.ws/sudo/intro.html
17. Miller, T.C.: Authentication bypass when clock is reset (2013). http://www.sudo.ws/sudo/alerts/epoch_ticket.html
18. Miller, T.C.: Potential bypass of tty_tickets constraints (2013). http://www.sudo.ws/sudo/alerts/tty_tickets.html
19. Miller, T.C.: Security policy bypass when env_reset is disabled (2014). http://www.sudo.ws/sudo/alerts/env_add.html
20. Napier, R.A.: Secure automation: achieving least privilege with SSH, Sudo, and Suid. In: Proceedings of the 18th USENIX Conference on System Administration (LISA), pp. 203–212. USENIX Association, Berkeley (2004)
21. O'Gorman, G., McDonald, G.: The elderwood project. Technical report, Symantec (2012)