Idea: Usable Platforms for Secure Programming – Mining Unix for Insight and Guidelines

Sven Türpe

Fraunhofer Institute for Secure Information Technology SIT, Darmstadt, Germany
sven.tuerpe@sit.fraunhofer.de

Abstract. Just as security mechanisms for end users need to be usable, programming platforms and APIs need to be usable for programmers. To date the security community has assembled large catalogs of dos and don'ts for programmers, but rather little guidance for the design of APIs that make secure programming easy and natural. Unix with its setuid mechanism lets us study usable security issues of programming platforms. Setuid allows certain programs to run with higher privileges than the user or process controlling them. Operating across a privilege boundary entails security obligations for the program. Obligations are known and documented, yet developers often fail to fulfill them. Using concepts and vocabulary from usable security and usability of notations theory, we can explain how the Unix platform provokes vulnerabilities in such programs. This analysis is a first step towards developing platform design guidelines to address human factors issues in secure programming.

1 Introduction

When humans, while interacting with technology, run into the same kind of problem often enough for us to see a pattern, the technology is often at fault: its design does not sufficiently take into account human factors and human capabilities. Programming is no exception, "programmers are people, too" [2]. We know numerous vulnerability patterns and collect them in databases like CWE (`http://cwe.mitre.org`), but the security community is only starting to pay attention to the human factors involved in secure programming and the usability of programming platforms [16], [6], [15], [4], [12].

A classical example of vulnerability-inducing platform design is the set-user-id/set-group-id (setuid/setgid) mechanism of Unix. Setuid lets Unix processes under certain conditions change their identity (persona) and thus their privileges, allowing users to run particular programs with elevated privileges.

While useful and even necessary sometimes, setuid is also an inexhaustible source of vulnerabilities. Hundreds of vulnerability reports related to setuid can be found in the U.S. National Vulnerability Database (NVD, `http://nvd.nist.gov`); new instances continue to appear [8]. While usability issues in the immediate setuid API have been addressed in the literature [5], [14], [7], setuid causes a cariety of challenges elsewhere, which are thus far only covered by secure programming guides [3], [10], [1].

This paper proposes to analyze the guidelines for writing secure setuid programs and the underlying API design from a usability perspective. While the secure coding guides are technically sound, programmers apparently have difficulties following them consistently. Are there features in the design of Unix APIs that make it hard to put secure coding advice into practice? If so, how could these APIs be improved to make it easier for programmers to write secure code?

The ultimate aim is a collection of applicable design principles for APIs and programming platforms that facilitate secure programming. Setuid and the Unix API constitute an ideal starting point for such an investigation: they have been widely deployed and used, so that a large body of accessible code exists. Setuid also facilitates comparison, as a flip of a bit changes the security context of a program without any change in its code.

After briefly describing the setuid mechanism, this paper applies parts of existing usability and usable security theory to a toy program, demonstrating how such an analysis can yield insight.

2 Setuid in a Nutshell

Unix processes access kernel functions and system resources through the kernel's syscall API. The kernel enforces two kinds of policies there. First, files and file-like resources (device files, named pipes, sockets, etc.) are subject to discretionary access control. Second, some functions in the syscall API require superuser privileges to be called at all or with unrestricted parameter values.

The kernel makes its access control decisions based on the persona associated with a process. A process persona comprises an effective user id (EUID) and one or more group IDs. File access control uses these IDs together with a file's ownership (user and group) to select the set of permission bits to evaluate before granting or denying access. The superuser (`root`, user ID 0) can override these permission checks and access any file. A process with effective user ID 0 is also the only way to get unrestricted access to the syscall API.

2.1 Setuid Mechanism

A regular process inherits its persona from its parent. This behavior corresponds with the intuition of a user session: upon login, a user obtains a shell process with the appropriate persona, and whatever is being run from there has the same persona and privileges [13], [9]. The setuid mechanism allows some programs to run with a different persona; it has two parts:

1. The setuid/setgid permission bits, when set on program files, override identity inheritance. When a program with one of these bits is executed, the child process runs with the effective user or group ID determined by the file owner, rather than those inherited from the parent. Inherited IDs are also preserved, so that the process can switch privileges.

2. The `setuid()`/`setgid(` family of system calls allows processes to manipulate their own persona, subject to a number of constraints. A process with EUID=0 can take on any persona; this is also used to configure the persona of a login shell after user authentication. Processes with other effective user IDs cannot normally change their persona. However, in conjunction with the setuid permission bits, a process can drop and regain privileges.

For detail on setuid, its pitfalls, and proposed design improvements see the literature on setuid [5], [14], [7] and Unix programming [10], [13].

2.2 Uses

Setuid is a versatile and useful mechanism and allows programs to handle cases not covered by the semantics and granularity of file access control. Setuid is used, for example, in these cases:

- The login program, running with root privileges, uses the `setuid()`/`setgid()` API to personalize the shell process for an authenticated user.
- File access control cannot enforce finer-grained policies. Unix password files, for example, need line-by-line access control so that non-`root` users can change only their own passwords. A setuid program can enforce arbitrary policies on resources accessible for the program but not for its users.
- Some programs need to make privileged system calls but should nevertheless be started and controlled by a regular user. The standard ports for HTTP (80) and HTTPS (443), for example, are privileged. Otherwise, however, a web server is a regular program that needs no special privileges.

In principle, setuid may be used with any user identity. However, setuid `root` is the most common and also the most critical use.

3 Security Obligations and Programming Rules

3.1 Security Obligations

Setuid places a process at a privilege boundary. Program code is being executed with elevated privileges while input is controlled and output is received by a user holding at most a subset of these privileges. Input includes a program's standard input stream, files read by the program, environment variables, signals, and possibly interactive commands. Output includes the standard output stream, error or log messages, and files written or manipulated.

On the one hand, setuid allows designated programs to refine and extend the access control policy enforced by a system. The `passwd` program, for example, which allows users to change their own password but not those of other users, enforces a policy on lines of the password file, whereas file access control can only enforce permissions on entire files. On the other hand, a program at a privilege boundary becomes a guardian of the higher privilege. A program meant to attain privileges by the setuid mechanism needs to make sure that

1. It enforces the required policy completely and correctly. Any failure to do so defeats its purpose.
2. No matter what the caller does to inputs and outputs, the program does not support any operation not part of its intended purpose.

The latter is the harder problem. Consider just some of the things that should not happen across a privilege boundary:

- Write user-controlled data to a user-selected file
- Execute user-specified commands or code with elevated privileges
- Read files and forward information about their content to the user.

Data and control flows must be carefully constrained across the entire input and output space of the program. Due to the purpose of the program – extending and refining access control – this burden rests with the program alone.

3.2 Programming Rules

The abstract obligations of a setuid program translate into a larger set of rules for the programmer. Bishop [3] developed an early set of rules, including items like:

- "Close all but necessary file descriptors before calling exec." (The exec call loads and runs a new program within the process, replacing the one currently running. Open files remain open.)
- "Check the environment in which the process will run." (The process environment is inherited from the parent. It contains a number of user-controlled variables and parameters, which influence the behavior of library functions and programs.)
- "Make only safe assumptions about recovery of errors." (Attempts at error recovery that might be helpful in a regular program can become dangerous in conjunction with setuid.)

Such rules have their roots in design subtleties that can be exploited in a setuid setting. The process environment, for example, is passed on silently in the background and controls critical behaviors of programs and libraries – how program files are searched, how new files are created, and so on.

Garfinkel et al. [10] later offered advanced design guidelines, advising programmers, for example, to bracket code sections that actually need elevated privileges between code that restores privileges before and drops them after a the respective calls. Chen et al. [5] propose a revision of the setuid API that makes this idiom easier to use and more robust.

4 Example: A Good Program Turning Vulnerable

4.1 Hello, World!

Listing 1 outlines a "Hello, world!" program, which instead of just printing its message, sends an email to the address specified as the first command line argument. After some declarations, the program creates a command string of the

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[]) {
5      char cmd[256] = "";
6      FILE *mail = NULL;
7      /* ... */
8      snprintf(cmd, 256, "mail %s", argv[1]);
9      mail = popen(cmd, "w");
10     fputs("Hello, world!\n", mail);
11     pclose(mail);
12     return 0;
13 }
```

Listing 1: This program has multiple vulnerabilities when executed setuid `root`.

form `mail <email_addr>` in the string buffer `cmd` (line 8), executes this command through a `popen()` call (line 9), and writes the message to the pipe thus opened (line 10). All error handling has been omitted for brevity, but should be straightforward: verify that `argv[1]` is present (let the `mail` program care about syntax) and check return values after each call.

Apart from the intentional omission of error handling, the program in Listing 1 exhibits two reasonable design decisions. First, the program reuses the existing `mail` command, which in turn takes care of the complexities of email sending. Second, our "Hello, world!" program uses `popen()` to call this subprogram. This, too, hides complexity from the programmer and takes care of many details. Using lower-level calls, such as `fork()` and `exec()`, to implement the same functionality would require a lot more code; `popen()` together with the file API of the C standard library offers a convenient abstraction. As `popen()` uses the Unix shell to execute commands and child processes inherit environments, programs executed this way also follow platform conventions, such as searching for programs as specified by the `PATH` variable or honoring the `LANG` and `LC_*` environment variables controlling internationalization.

4.2 Some Vulnerabilities

As soon as the setuid mechanism is applied to run the program in Listing 1 with `root` privileges while keeping an unprivileged user in control of its inputs, the program becomes a ragbag of vulnerabilities:

- Line 8 embeds user input in a command string to be executed by a command interpreter. The caller can use a variety of separators and other mechanisms to sneak in arbitrary commands to be executed with elevated privileges.
- Line 8 does not specify an absolute path for the `mail` command. The shell invoked by the `popen()` call in line 9 will hence search for an executable file named `mail` in the directories specified in the `PATH` environment variable; the

caller can manipulate the search path so that an arbitrary program named `mail` is found first.
- Line 9 executes the prepared command as a child process, passing on all environment variables. These variables may influence the operation of the mail command or the hidden shell used to execute this command.

As an immediate mitigation, the programmer might (1) specify an absolute path to the mail command [10] and (2) use lower-level calls to spawn the `mail` subprocess without executing a shell command or searching files along an environment-specified path [1]. It is also recommended to (3) sanitize environment variables [3], [10] and to (4) assure open file descriptors do not leak across privilege boundaries [3].

To reduce the risk from buffer overflow and similar defects, which can occur anywhere in a program, and as a general matter of hygiene, it is further recommended to employ either of two patterns dependent on how often elevated privileges are required: (a) carry out all privileged work early, then drop privileges permanently, or (b) drop privileges temporarily whenever they are not needed. This reduces the amount of code actually running with elevated privileges. Getting privilege changes right can be a challenge of its own [5], [14].

5 The API Usability Perspective

For every possible functional specification we can think of two different programming tasks T_R and T_S, where T_R is the task of writing a program that approximates the specified behavior closely enough, and T_S is the same task with the additional requirement that the result also be secure. Translated into Unix with its setuid mechanism, T_R is the task of writing a regular program that runs with the privileges of the user controlling its inputs, and T_S is the task of writing a functionally equivalent program suitable for setuid `root` use.

Ideally, task T_S of writing a secure program should not be harder to accomplish than task T_R of writing an equivalent regular program. If we can identify factors in platform and API design that systematically make T_S harder to accomplish than the corresponding T_R, then the platform leaves room for usability improvement. The ideal "don't care" situation may not be attainable, but perhaps security mechanisms and APIs can be redesigned to make it easier for programmers to fulfill their remaining security duties.

To identify factors that complicate secure programming we can apply usable security principles [17] and general API usability guidelines [11]. The following two subsections will illustrate this for subsets of the respective criteria.

5.1 Usable Security Principles

Yee [17] proposes ten principles of user interaction design for secure systems. Although programming tasks differ in important respects from interactive use of a program or security mechanism, some of these principles can be applied to programming environments. Two examples:

Path of Least Resistance. "The most natural way to do any task should also be the most secure way" [17]. This is a different way of putting the ideal outlined above, where the programmer just does not have to care about security. Many secure programming rules imply that programmers should replace short and straightforward pieces of code with longer and more complicated ones. Apple's secure programming guide [1], for example describes a supposedly secure alternative to the `popen()` call. This recommended alternative would vastly increase the length of the example in Listing 1, introduce some potential for new defects, and require the programmer to deal with lower-level APIs. Requiring such programming games [11] clearly violates the path of least resistance principle.

Explicit Authorization. Originally referring to transfers of a user's authority to others, explicit authorization can be required for any critical aspect. The setuid mechanism violates this principle by placing programs in a security-critical context without asking for the programmer's consent. Rather than letting the programmer acquire privileges when needed, the platform forces programmers to drop privileges when they do not need them. From the programmer's point of view, running with elevated privileges is the default rather than an explicitly authorized exception.

5.2 Cognitive Dimensions

The cognitive dimensions framework [11] provides a vocabulary to discuss usability properties of programming languages, APIs, and other information artefacts. A discussion by the cognitive dimensions merely describes properties of a notation; how these properties affect usability depends on the kind of task to be accomplished. Programming as an interactive design task imposes high demands on the notation. The cognitive dimensions include aspects like the following:

Hard Mental Operations. Some operations, such as Boolean logic, are inherently hard to carry out for the human mind. A notation requiring such operations to be understood therefore becomes hard to use. Secure programming in a setuid scenario requires the programmer to keep track of data flows between the two privilege levels, regular and elevated, and make sure the program cannot be abused to read or write data with elevated privileges beyond its intended purpose and policy. However, this is nearly impossible even for small programs. The example in Listing 1 is only a toy program without subroutines, yet it contains already two indirections: to understand the `fputs()` call in line 10, one has to track the file handle mail to the `popen()` call before, which in turn depends on a command assembled in line 8 using user input (`argv[1]`). This becomes hopeless rather quickly as programs grow.

Visibility. To write secure setuid programs, the programmer has to follow numerous rules, but the API does not give any hints as to which rules to apply

when and where. There are no defined markers for safe or unsafe functions or for data that could or should not be used in certain ways. Secure programming rules rely entirely on information in the programmer's head. The programmer needs to know the inner workings of functions like `popen()` to understand the risks, contrary to the idea that functions should hide implementation detail and rather adhere to an explicit contract.

Hidden Dependencies. Hidden dependencies occur if actions in one place have a non-obvious effect elsewhere. In the case of `popen()`, a hidden dependency exists between the process environment and the behavior of `popen()`. Environment variables are being passed on down the process tree. The programmer can intervene, but as a default, environment variables are hidden from the programmer rather than passed explicitly.

6 Outlook

Although incomplete, the preceding analysis already suggests some directions for API redesign. An improved version of the Unix API could for example:

- Let programmers acquire privileges and corresponding responsibilities through an explicit call,
- Offer safe alternatives to unsafe functions, so that secure alternatives do not require writing more code, and
- Detect inappropriate security contexts inside critical functions and return an error when a functions is being called where it shouldn't.

The second step of research after analysis of the existing API is therefore improvement. Tradeoffs will likely appear between the different design goals, so even if we know what to aim for, devising an improved API remains a challenge. Finally, any proposed improvement needs to be tested with real programmers. This may be the hardest part. Research prefers small, controlled lab experiments, whereas real programming takes place in large projects and code bases and is done by programmers that acquire skills and habits over time as they use and reuse platforms. As an alternative, once a set of usability principles has been established, other platforms and their program vulnerbility patterns can be analyzed to see whether the principles explain the patterns.

For the first two steps, a vast amount of data is freely available for research. Vulnerability databases are full of reports of defect instances. Many of those concern open source software and can be reviewed. Open source platform implementations – Linux and *BSD – facilitate experimentation, the more so as extensions like Linux capabilities, SELinux, and Capsicum exist, which address the same set of issues from a technical rather than from a human factors perspective.

References

1. Apple Inc.: Secure Coding Guide, 2014-02-11 edn. (2006-2014), https://developer.apple.com/library/mac/documentation/Security/Conceptual/SecureCodingGuide/
2. Arnold, K.: Programmers are people, too. ACM Queue 3(5), 54–59 (2005)
3. Bishop, M.: How to write a setuid program. ;login: 12(1), 5–11 (1987)
4. Cappos, J., Zhuang, Y., Oliveira, D., Rosenthal, M., Yeh, K.C.: Vulnerabilities as blind spots in developer's heuristic-based decision-making processes. In: Proc. New Security Paradigms Workshop. pp. 53–62. NSPW '14, ACM, New York, NY, USA (2014)
5. Chen, H., Wagner, D., Dean, D.: Setuid demystified. In: USENIX Security Symposium. pp. 171–190 (2002)
6. Crandall, J.R., Oliveira, D.: Holographic vulnerability studies: vulnerabilities as fractures in interpretation as information flows across abstraction boundaries. In: Proc. New security paradigms workshop. pp. 141–152. NSPW '12, ACM, New York, NY, USA (2012)
7. Dittmer, M.S., Tripunitara, M.V.: The unix process identity crisis: A standards-driven approach to setuid. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 1391–1402. CCS '14, ACM, New York, NY, USA (2014)
8. Esser, S.: OS X 10.10 DYLD_PRINT_TO_FILE local privilege escalation vulnerability. https://www.sektioneins.de/blog/15-07-07-dyld_print_to_file_lpe.html (Jul 2015)
9. Free Software Foundation, Inc.: The GNU C Library Reference Manual, glibc 2.22 edn. (Aug 2015), https://www.gnu.org/software/libc/manual/
10. Garfinkel, S., Spafford, G., Schwartz, A.: Practical UNIX and Internet Security. O'Reilly Media, 3rd edn. (2003)
11. Green, T.R.G., Petre, M.: Usability analysis of visual programming environments: A 'cognitive dimensions' framework. Journal of Visual Languages & Computing 7(2), 131 – 174 (1996)
12. Oliveira, D., Rosenthal, M., Morin, N., Yeh, K.C., Cappos, J., Zhuang, Y.: It's the psychology stupid: How heuristics explain software vulnerabilities and how priming can illuminate developer's blind spots. In: Proc. 30th Annual Computer Security Applications Conf. pp. 296–305. ACSAC '14, ACM, New York, NY, USA (2014)
13. Stevens, W.R.: Advanced Programming in the UNIX Environment. Addison-Wesley Publishing Company (1992)
14. Tsafrir, D., Da Silva, D., Wagner, D.: The murky issue of changing process identity: revising "setuid demystified". ;login: 33(3), 55–66 (2008)
15. Türpe, S.: Point-and-shoot security design: Can we build better tools for developers? In: Proc. New Security Paradigms Workshop. pp. 27–42. NSPW'12, ACM, New York, NY, USA (2012)
16. Wurster, G., van Oorschot, P.C.: The developer is the enemy. In: Proc. New security paradigms workshop. pp. 89–97. NSPW '08, ACM, New York, NY, USA (2008)
17. Yee, K.P.: User interaction design for secure systems. In: Deng, R., Bao, F., Zhou, J., Qing, S. (eds.) Information and Communications Security, LNCS, vol. 2513, pp. 278–290. Springer Berlin / Heidelberg (2002), 10.1007/3-540-36159-6_24