University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Paweł Hajdan
Student no. 262531

More Secure Implementation of UNIX Shadow Utilities

Master's thesis
in COMPUTER SCIENCE

Supervised by
Janina Mincer-Daszkiewicz, Ph. D.
Institute of Computer Science,
University of Warsaw, Poland

June 2012

Supervisor's statement

Hereby I confirm that the present thesis was prepared under my supervision and that it fulfills the requirements for the degree of Master in Computer Science.

Date Supervisor's signature

Author's statement

Hereby I declare that the present thesis was prepared by me and none of its contents was obtained by means that are against the law. The thesis has never before been a subject of any procedure of obtaining academic degree. Moreover, I declare that the present version of the thesis is identical with the attached electronic version.

Date Author's signature

Abstract

In this thesis a new implementation of UNIX shadow utilities is presented: hardened-shadow. Compared to existing account management packages, it applies principle of least privilege more thorougly, and provides increased security without need to patch other system components (some configuration changes are needed though). Care has been taken to follow state-of-the-art secure coding guidelines. The code has been published on the internet as an Open Source project.

Keywords

security, password shadowing, password aging, password security, user account management, privilege escalation, SUID, SGID, UNIX, Linux, shadow-utils, tcb, Openwall

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Computer Science

Subject classification

D. Software
D.4 Operating Systems
D.4.6 Security and Protection
D.4.9 Systems Programs and Utilities

Title in Polish

Bezpieczniejsza implementacja uniksowego pakietu shadow

Contents

List of Tables

Chapter 1

Introduction

1.1. Computer (in)security

Software is becoming more and more important part of critical infrastructure, i.e. assets essential for the society and economy like telecommunication, transportation, watter supply, energy and financial services. This popularity and broad usage in networked environments creates new challenges is securing these software solutions. Contrary to the common belief, the most dangerous adversaries are not self-taught rebellious teenagers, but nation-state intelligence agencies engaging in espionage and organized criminals trying to make money. It is especially important to secure the foundation of our digital infrastructure, i.e. hardware and operating systems. Higher-level solutions are also needed, but when the low-level components are vulnerable, it is easy to bypass security checks in higher layers. Breaking into administrator's account and replacing application binaries makes any security measures in the application itself completely ineffective.

1.2. User account security

One of the key ideas of multi-user operating systems is isolating the users from each other. Authentication is then needed so that the user's identity can be verified. Most often passwords are used for this purpose, and even when some other way of authentication is used (e.g. cryptographic keys), it is rare to disable passwords completely. Even when remote password-based login is disabled, at least local infrastructure for managing passwords exists.

1.2.1. General password security measures

Passwords may be lost, stolen or guessed. To protect against consequences of password loss, organizations often require periodical change of passwords, i.e. password aging also referred to as password expiry. If an attacker obtains a password, he can use it up to the next change. Weak passwords can be guessed or brute-forced, so a typical password policy also has some requirements on password complexity. To protect against brute-force attacks, the system may lock out accounts after too many failed login attempts (which by the way may introduce a Denial-of-Service vulnerability).

Password policies do not solve all problems though, like users writing down passwords or choosing new passwords in a predictable pattern.

1.2.2. Password hashing

Passwords need to be stored somehow, in a way that makes them hard to steal but easy to match against password provided during login. Most common solution is using a cryptographic hash function and only store the hash value. It is computationally expensive and infeasible in practice to obtain the original password based on its hash.

However, an attacker can pre-compute hashes for certain classes of weak passwords like short ones not using special characters, or dictionary-based passwords. The tables of pre-computed hashes used for password cracking are called rainbow tables.

To defend against rainbow tables, password salting can be used. This means that for each password a random salt is generated, and stored in clear text. When computing the hash, the salt and password are concatenated and the result is hashed. This way, two different users with same password would have different hashes because of different random salts. The attacker would need a different set of rainbow tables depending on the salt used, making that approach infeasible.

A notable hash function for passwords is bcrypt [8], designed by Niels Provos and David Mazières. It is based on Blowfish cipher, and is an adaptive hash, which means it can be made more and more computationally expensive (i.e. slower) as the computers become faster.

Another password hash function is PBKDF2 (Password-Based Key Derivation Function). It uses a salt and multiple iterations to make the function more resistant against brute-force attacks. PBDKF2 is used as part of wireless networks encryption protocols and full disk encryption software, but I have not found it being used to hash account passwords on UNIX systems.

1.2.3. UNIX password security

Traditionally UNIX systems use /etc/passwd file to list user accounts in the system and /etc/-group to list user groups in the system. Both of these files are world-readable, i.e. any user on the system can read them. In the early days of UNIX, /etc/passwd contained the password hashes. Later it became a common hacking technique to get access to the system's passwd file, crack the passwords using a fast machine (the hash was not particularly strong), and obtain administrator's password. To defend against that, password shadowing has been introduced. Hashes have been moved to /etc/shadow file, readable only be the administrator (root). This way regular users do not have access to the hashes. However, system utilities that manipulate passwords, still need to run with elevated privileges: users need to change passwords, and /etc/shadow file (which only administrator can read and edit) needs to be modified to do that.

At some point group passwords have been added: user could use newgrp to add more groups to his supplementary group set, provided he knew passwords for these groups. The idea seems rather outdated now, e.g. because it is difficult to remove people from password-protected groups (the password has to be changed, which is inconvenient). It also adds complexity to the management software, and also increases attack surface (newgrp has to run with elevated privileges), but is still supported by mainstream packages for compatibility reasons.

1.3. Confinement of user-space programs on UNIX

1.3.1. Discretionary access control and privilege elevation

Traditionally, UNIX systems use discretionary access control, which means that a subject that has certain permission can pass that permission to any other subject. In practice this means a program running as a certain user has full privileges of that user.

UNIX also introduced the SUID and SGID file access rights, which allow users to run programs with privileges of the executable owner or group owner, respectively. Among others, password manipulation program "passwd" has SUID root bit set, because ordinary users obviously cannot manipulate passwords file directly. The "passwd" program is supposed to only change a single user's password after authenticating that user. However, because of discretionary access control, it runs with full privileges of the root user, which makes it a target for privilege escalation attack. When "passwd" (or any other SUID root program) is compromised, the attacker gains full administrative access to the system.

1.3.2. Mandatory access control

To protect against certain classes of attacks, and increase security in general, mandatory access control has been invented. It introduces a centrally managed security policy that cannot be overriden by the users. It allows confining different processes even if they run as the same user.

A notable mandatory access control implementation is SELinux, which consists of both user- and kernel-space components. It is commonly used in production environments, and has successfully mitigated several vulnerabilities, e.g. in samba[1], Mambo[2], and Linux kernel[3].

It might be tempting to ignore security of individual applications and just use SELinux to confine them. This is not really a good approach, as SELinux alone cannot stop all vulnerabilities, especially in the kernel. A notable example is Linux Cheddar Bay privilege escalation exploit[4] from 2009, which works even when SELinux is enabled, and as part of the payload disables SELinux and other security features. This exploit does not render SELinux completely useless of course. We still need security by isolation, as stressed[5] by Joanna Rutkowska. The main point here is that applications should not rely on SELinux for security.

1.4. Existing password shadowing solutions

Password shadowing is not a new concept, so there are different solutions already in use. The packages listed below are the most popular ones. The list is not necessarily exhaustive. hardened-shadow described in this thesis is expected to have at least one major advantage over each of the following solutions.

1.4.1. shadow-utils

Originally known as "Shadow Password Suite" the package has been written in 1987 by Julie Haugh. In 1992 it has been ported to Linux (which was still early in development).

[1] http://danwalsh.livejournal.com/10131.html
[2] http://www.linuxjournal.com/article/9176
[3] https://www.redhat.com/archives/fedora-selinux-list/2006-July/msg00071.html
[4] http://lwn.net/Articles/341773/
[5] http://theinvisiblethings.blogspot.com/2008/09/three-approaches-to-computer-security.html

Since the beginning security problems in the shadow packages have been critical, including buffer overflow in login[6] discovered by Joseph Zbiciak in 1996. It appears that later developers of PLD Linux Distributions, mainly Marek Michałkiewicz, Tomasz Kłoczko took over maintenance of the code. Since 2007 the project is maintained by Debian Linux developers.

It seems to be the most popular implementation in use. It is used at least by Debian, Ubuntu, Mandriva, RedHat, Fedora, Gentoo, Arch.

1.4.2. pwdutils

There is an alternative implementation called pwdutils, written mainly by Thorsten Kukuk (earliest change log entry is from 2002) and maintained as part of Linux-NIS project. It is used at least by SuSE Linux.

1.4.3. SimplePAMApps

Another alternative implementation using Linux-PAM started in 1997 by Andrew Morgan. Currently it does not seem to be in active development. Does not have an actual webpage, but is still in active use, at least by Openwall GNU/Linux and ALT Linux (with custom patches).

1.4.4. tcb

Rather an add-on than full implementation, Openwall's tcb makes it possible to run shadow utilities with lower privileges (SGID instead of SUID). It contains a library, an NSSwitch module and a PAM module, and requires patching the shadow suite in use. The C library also has to be patched to support Openwall crypt(3) extensions. It is used by Openwall GNU/Linux, but has not gained mainstream adoption likely because of the need for custom-patching glibc and shadow.

It has directly inspired hardened-shadow, which uses a very similar directory structure and shares many goals with tcb. The main difference is that hardened-shadow is a complete solution, which contains programs, NSSwitch and PAM module all integrated in a single package, with no patching required.

1.5. Project goals

The goal of the project is to design and implement a new password shadowing package that can be used on a Linux system, which offers significantly stronger security compared to existing solutions by reducing attack surface, applying the limited privilege principle more thoroughly, and using secure coding guidelines. Additionally, the code is going to be made available for free on the internet as an open source project, to get feedback about its usefulness and security. Usage in real-world scenarios is the best way to verify whether stated goals have been achieved.

1.6. Structure of the thesis

The Glossary (Chapter 2) contains definitions of various terms used in this thesis. In Chapter 3 design requirements are listed and explained. In Chapter 4 it is described how hardened-

[6] http://www.redhat.com/archives/linux-security/1996-December/msg00020.html

shadow works in practice, and what are its distinguishing features. In Chapter 5 some implementation details are discussed. In Chapter 6 everything related to open-sourcing the code is presented. Finally, in Chapter 7 a summary of the accomplishments is provided.

Appendix A contains information about CD attached to this thesis and detailed instructions how to build and install hardened-shadow. Appendix B contains a full text of the software license that applies to hardened-shadow. Appendices C and D contain notable code snippets from hardened-shadow, that would be too long to include inline in earlier chapters.

Chapter 2

Glossary

This glossary explains some technical terms used in this thesis. The reader is likely familiar with most of them, but the definitions are provided for completeness and to avoid possible ambiguity.

arbitrary code execution attacker's ability to run code of his choice with the privileges of the vulnerable process

ASLR also known as Address Space Layout Randomization, a security hardening technique which involves randomly arranging positions of logical areas in a process's address space

brute-force attack an attack that is based on exhaustive search of all or a subset of possible inputs, like passwords or cryptographic keys

BSD license one of permissive free software licences; places only minimal restrictions on how the software can be distributed, and does not have any reciprocity or share-alike requirements

buffer overflow violation of memory safety which causes the program to overrun a buffer's boundary and overwrite adjacent memory; it can lead to arbitrary code execution

CERT Computer Emergency Response Team, an expert group that handles computer security incidents

Denial-of-Service attack type of attack that results in a service being unavailable to its intended users

DocBook XML-based language for technical documentation

GNU UNIX-like operating system; recursive acronym for "GNU's Not UNIX!"

GPG GNU Privacy Guard, suite of cryptographic software

GPL GNU General Public License, a copyleft free software license, which requires the derived works to be distributed under the same terms

m4 macro processing language designed by Brian Kernighan and Dennis Ritchie, commonly present on UNIX systems

mandatory access control type of access control where security policy is managed centrally by an administrator and users cannot override the policy (e.g. by granting access that is denied by the policy)

memory safety correctness of memory access meaning that offset and length of every read or write always refers to intended logical object

NIS also known as Network Information Service or Yellow Pages, protocol for distributing system configuration data on a network

NSSwitch also known as Name Service Switch, an operating system facility that makes it possible to configure different sources of common system databases like user accounts and passwords

OpenBSD UNIX-like operating system focused on security and code correctness

PAM also known as Pluggable Authentication Modules, a mechanism to integrate multiple authentication schemes into a high level, configurable interface that applications can use; this allows easy change of authentication scheme without changing applications

patching software applying custom modifications to software

PIC also known as Position Independent Code, executes properly regardless of the address it is placed at in memory, without modification

PIE also known as Position Independent Executable, an executable binary made entirely from position-independent code, making it possible to use ASLR

principle of least privilege requirement that every piece of software or a user operates with the least amount of privilege necessary for its job

shell command-line interpreter

SELinux also known as Security-Enhanced Linux, set of Linux kernel features and userspace tools that implement mandatory access control

SGID UNIX access rights flag which allows users to run an executable with privileges of the executable group owner's (set-group-ID)

SUID UNIX access rights flag which allows users to run an executable with privileges of the executable owner's (set-user-ID)

tarball an archive created with tar(1)

vulnerability flaw in system design or implementation that could be exploited and result in a security breach

XML Extensible Markup Language, a markup language designed for both human- and machine-readability of encoded documents

Chapter 3

Design requirements

Well thought-out design is important for high-quality software. The input for the design process are requirements. In case of hardened-shadow, the design requirements are based on analysis of existing password shadowing suites' strenghts and weaknesses, and expectations of the target audience: system administrators and operating system distribution packagers.

3.1. Small set of C programs

System administrators expect the user management utilities to have command-line interface similar to what they are familiar with. Generally this kind of software is written in C, and using any other language including C++ would likely decrease the adoption rate of the implementation. Here is complete list of needed executables (the descriptions mostly come from man pages):

chage change user password expiry information

chsh change login shell

groupadd create a new group

groupdel delete a group

groupmod modify a group definition on the system

groups display current group names

grpck verify integrity of group file

lastlog report the most recent login of all users or of a given user

login begin session on the system

nologin politely refuse a login

passwd change user password

pwck verify integrity of password files

pwconv convert to hardened-shadow passwords

pwunconv convert from hardened-shadow passwords

shell_proxy launch user's chosen shell

su change user ID or become superuser

useradd create a new user or update default new user information

userdel delete a user account and related files

usermod modify a user account

vipw edit the password or group file

3.2. No need for custom patching

The whole suite should be self-contained, and should not require non-standard modifications to other parts of the system, like the C library, or programs in other packages.

Need for custom patching discourages Linux distributions from packaging the software, and system administrators from trying it (for maintainability reasons).

3.3. GNU build system

Autotools (autoconf, automake, libtool) is the de-facto standard for packaging C software for UNIX. It is most widely supported, and the most Linux distribution packagers know how to handle it. Is is worth to note that the build is non-recursive, which is considered better than recursive make build [7]. PIE (Position Independent Executable) and PIC (Position Independent Code) are enabled for all code. The performance hit is negligible (hardened-shadow code is not performance-critical anyway), and the benefits of exploit mitigation techniques for security-critical code outweigh any possible downsides.

3.4. Limited privileges

The only SUID program must be su. Other programs such as passwd should be at most SGID, to limit possible damage in case the program is compromised. This is not an easy requirement to achieve. Among existing solutions, only tcb runs with limited privileges, but requires modification to other system components. One of the main advantages of hardened-shadow is providing the increased security in a single, integrated package.

3.5. Simple codebase

Security-critical software must be easy to audit. The amount of code should be limited, and the implementation itself should be straightforward, easy to read and analyse. Examples of constructs to be avoided are:

- pre-processor directives like #ifdef for conditional features or compatibility,
- function pointers,
- lots of global state,
- low-level code where better abstractions can be used,

- more portable code at the expense of readability,
- non-trivial error handling code,
- inconsistencies in error handling,
- non-descriptive, generic function names,
- excessive logging, cluttering the code,
- excessive annotations for static-analysis tools,
- long comments that add nothing to the code itself.

3.6. Compromises

The priorities of the project are as follows. In case of conflicting goals, higher priority property is favored over lower priority one:

1. security,
2. readability,
3. correctness,
4. maintainability,
5. efficiency,
6. features.

Account management software is security-critical but not performance-critical, so efficiency near the end of the list should not be surprising. I favor readability over correctness, which means handling of obscure edge cases that do not matter for security but e.g. user friendliness may be skipped if they result in ugly code or even too much additional code. Features may be dropped if that helps implement the core functionality in a secure manner on time. Example of features that are fine to drop (compared to existing alternatives) are support for NIS, SELinux, auditd, and group passwords.

Chapter 4

How it works

In this chapter, the distinguishing features of hardened-shadow are described: how the suite works in practice, from the perspective of a system administrator. Some design and implementation details that are important for better understanding of the user-visible interface are also described.

4.1. Executable permissions

Table 4.1: UNIX permissions of executables that are part of a shadow suite and run with elevated privileges

executable	traditional		hardened-shadow	
su	`-rws--x--x`	`root:root`	`-rws--x---`	`root:wheel`
passwd	`-rws--x--x`	`root:root`	`-rwx--s--x`	`root:hardened-shadow`
chage	`-rws--x--x`	`root:root`	`-rwx--s--x`	`root:hardened-shadow`
chsh	`-rws--x--x`	`root:root`	`-rwx--s--x`	`root:hardened-shadow`
chfn	`-rws--x--x`	`root:root`		`n/a`
expiry	`-rws--x--x`	`root:root`		`n/a`
gpasswd	`-rws--x--x`	`root:root`		`n/a`
newgrp	`-rws--x--x`	`root:root`		`n/a`
get_shell	`n/a`		`-rwx--s--x`	`root:hardened-shadow`

Note how hardened-shadow reduces attack surface:

1. su is only executable by the wheel group, which helps prevent privilege escalation attacks
2. passwd, chage and chsh are SGID instead of SUID root — i.e. run with lower privileges
3. chfn, expiry, gpasswd and newgrp are not shipped at all

4.2. Directory structure

Instead of a single /etc/shadow file hardened-shadow maintains /etc/hardened-shadow directory tree. Each user has its own subdirectory, which contains three files: shadow, aging, and shell. The user subdirectories are owned by respective users, and group-owned by hardened-shadow group.

```
drwxr-x--- 178 root   hardened-shadow 4096 /etc/hardened-shadow
drwx--x---   2 nobody hardened-shadow 4096 /etc/hardened-shadow/nobody/
-rw-r-----   1 nobody hardened-shadow   15 /etc/hardened-shadow/nobody/aging
-rw-r-----   1 nobody hardened-shadow    8 /etc/hardened-shadow/nobody/shadow
-rw-r-----   1 nobody hardened-shadow   10 /etc/hardened-shadow/nobody/shell
drwx--x---   2 root   hardened-shadow 4096 /etc/hardened-shadow/root
-rw-r-----   1 root   hardened-shadow   15 /etc/hardened-shadow/root/aging
-rw-r-----   1 root   hardened-shadow  106 /etc/hardened-shadow/root/shadow
-rw-r-----   1 root   hardened-shadow    9 /etc/hardened-shadow/root/shell
```

It is critically important that only members of hardened-shadow group have access to /etc/hardened-shadow directory. This ensures that any changes to these important files goes through trusted executables. Regular users must not have direct access to the files: that would allow them to change password without obeying password complexity policy and change frequency constraints, and also by manipulating the aging file to unlock the account or violate password aging policy.

It was quite a challenge to satisfy above requirement and keep all programs except su running with only SGID privileges. Especially the shell handling proved to be difficult and error-prone. In fact, during development many bugs have been found and fixed in that area.

The shadow file contains password hash and date of last password change:

```
$6$yncNA5B3ye4AUFgo$lY/4FsM6h5h9IvPBm84l43dfS9klEbOmlo6ec3TB5JMem9vBS6jFE/
caYePvVUQUBeiJ8qbbR6gpz48HTj8FI0:15498:
```

Note that `$6$` indicates the hash type (SHA-512 in this case). `yncNA5B3ye4AUFgo` is the salt. hardened-shadow uses maximum permissible 16 characters for the salt, instead of the default 8. This additional entropy provides better protection against brute-force attacks.

The aging file contains data about password aging: minimum and maximum number of days between password change, number of days before password expires to warn the user, number of days after password expires until account is disabled, and date when account expires (if applicable):

```
0:99999:7:-1:-1:
```

Note how -1 is used instead of an empty string, and the semicolon at the end to make parsing slightly easier.

The shell file contains path of the user's shell (that makes it possible to remove SUID bit from chsh):

```
/bin/bash
```

Note that there is no newline at the end of the shell file.

4.3. Changing user accounts

A common operation on the user files in /etc/hardened-shadow tree is replacement, which means creating a temporary file, writing its entire contents, and calling renameat(2) followed by fsync(2) on the parent directory descriptor. This ensures that the file rename has reached the disk. The routine to do that is called hardened_shadow_replace_user_file.

Password change is just replacing the user's shadow file with a new one, which has updated password hash and date of last change.

Similarly, shell change is replacing the user's shell file with path of the new shell. To maintain full compatibility with programs that read /etc/passwd directly, /bin/shell_proxy is introduced and used as a shell in /etc/passwd. shell_proxy reads the user's shell file in /etc/hardened-shadow (using privileged get_shell executable), adjusts the environment as if the target shell was called, and finally calls execv(3). The environment change means the following: SHELL environment variable is set to the target shell, and argv[0] is changed to the name of the target shell. Note that when argv[0] begins with a hyphen (a convention that apparently means a login shell), it is preserved (i.e. target shell is used with a hyphen prepended).

4.4. Configuration

Obviously not all systems are alike, and some degree of configuration is required. hardened-shadow deliberately keeps the configurable settings to a minimum, and the defaults should be fine for most systems. See table 4.2 for the complete list of supported settings.

Table 4.2: Configuration settings supported by hardened-shadow

name	type	description	default
CREATE_HOME	boolean	whether to create home directory when adding a user	yes
USER_PRIVATE_GROUPS	boolean	whether to create a group for each user	yes
PASS_MIN_DAYS	integer	default minimum number of days between password changes	0
PASS_MAX_DAYS	integer	default maximum number of days between password changes	99999
PASS_WARN_AGE	integer	default number of days before password expires to warn the user	7
HOME_DIRECTORY_MODE	mode	UNIX access rights for the home directory	0755
MAIL_DIRECTORY	path	where mail spool is located	/var/spool/mail
USERDEL_COMMAND	path	command executed on user deletion; should remove user's cron jobs etc	/bin/true
USER_UID_RANGE	range	valid non-system UIDs	1000-60000
SYSTEM_UID_RANGE	range	valid system UIDs	101-999
USER_GID_RANGE	range	valid non-system GIDs	1000-60000
SYSTEM_GID_RANGE	range	valid system UIDs	101-999

4.5. Miscellanous

Care has been taken to avoid file descriptor leaks on new process launch. O_CLOEXEC flag is the best way to accomplish that, but as the last line of defense (and with libraries we cannot be sure, e.g. glibc, PAM) all possible file descriptors are closed (hardened_shadow_closefrom). It is interesting that e.g. OpenBSD has a very similar function built-in — closefrom(2), but Linux does not. There are concerns[1] that a Linux implementation of closefrom (in fact hardened-shadow uses a very similar approach) is not async-signal-safe. Note however that according to OpenBSD's signal(3) man page, OpenBSD's closefrom is not async-signal-safe either.

Data about password expiry has been put in the aging file, separate from the shadow one. This way the user password can be changed atomically without need to touch or lock the aging information. Note that the aging file is not normally available to users. Only hardened-shadow programs can access it. Of course the programs could be exploited to do something malicious with the aging information (including modification), but it is also true for traditional shadow file layout.

[1] http://lwn.net/Articles/293459/

Chapter 5

Implementation considerations

In this chapter implementation details that do not directly influence the user-visible parts of the suite are described. Even though they are more "internal", they are very important from a developer's point of view: they contribute to the security and readability of the hardened-shadow's codebase.

5.1. CERT C Secure C Coding Standard

While working on hardened-shadow I was frequently referring to [9], and here I would summarize some guidelines that are especially worth noting.

5.1.1. INT04-C. Enforce limits on integer values originating from untrusted sources

This is built into very design of hardened-shadow, with functions like hardened_shadow_strtonum (inspired by strtonum(3) from OpenBSD), hardened_shadow_getrange and implementation of configuration file parsing. The goal is to make it easy and natural to follow this rule throughout the codebase. In fact, the interface of functions mentioned above results in correct code even if one does not explicitly know this rule, but just uses the right functions.

5.1.2. INT30-C. Ensure that unsigned integer operations do not wrap

Unsigned integer operations are often used to calculate buffer sizes. Integer-related vulnerabilities can often lead to memory corruption vulnerabilities, so it is important to avoid them. I chose a comprehensive approach like above, and implemented easy to use checks like hardened_shadow_uadd_ok, hardened_shadow_umul_ok and so on — the examples listed here check whether unsigned addition and multiplication are safe to perform.

Using C99's [5] uintmax_t type, which is guaranteed to be the largest unsigned integer type to implement the above functions helps prevent even more mistakes with mismatched types.

Implementation of the unsigned integer checks can be reviewed in Appendix C.

5.1.3. INT32-C. Ensure that operations on signed integers do not result in overflow

Similarly to the above, I used functions like hardened_shadow_sadd_ok to check for overflow when operating on signed integers. An alternative approach would be to rely on GNU GCC's

-fwrapv or -ftrapv options (which make integers wrap around to zero or abort the program respectively in case of an overflow). However, using functions allows full control over program behavior, and graceful handling of failure.

Implementation of the signed integer checks can be reviewed in Appendix C.

5.1.4. MEM01-C. Store a new value in pointers immediately after free()

Probably not all places in the code follow that guideline, but it is a very useful one. It is not obvious from the name of this guideline, but the new value that should be stored in pointers is NULL. Debugging a NULL pointer dereference is so much easier than a strange crash caused by memory corruption. Also, crashing on NULL is generally not exploitable, while crashing on a non-NULL pointer often can be easily exploited.

5.1.5. ENV03-C. Sanitize the environment when invoking external programs

This is important especially for su. hardened-shadow creates an empty environment that is later populated with standard variables like HOME (selected variables can be preserved, if needed).

The implementation can be reviewed in Appendix D.

5.1.6. SIG02-C. Avoid using signals to implement normal functionality

Sometimes signals are used to handle login timeouts and similar events. hardened-shadow does not use any custom signal handlers. The only thing it does with signals is resetting signal handlers to default. Signal-related races are especially dangerous when running with elevated privileges, even temporarily. For example, in 1997 a serious security bug[1] has been found in wu-ftpd.

5.1.7. SIG30-C. Call only asynchronous-safe functions within signal handlers

While hardened-shadow does not use signal handlers, I would like to note here that what can be safely done in these handlers is very limited. The safest thing to do is to set a flag later read from the main program. The handler cannot allocate memory on heap, cannot use printf, and so on.

5.1.8. ERR02-C. Avoid in-band error indicators

It is common to return "nonsensical" or "impossible" values like -1, empty string or NULL to indicate failure. Interfaces designed that way are very error prone. It is easy to skip error checking, and it is also easy to use the returned value as if it was valid, e.g. dereferencing NULL pointer or — more dangerously — performing arithmetic on the returned -1 value (which is assumed to be positive), leading to unexpected results. hardened-shadow uses explicit boolean success return values where possible and practical, and everything else is returned via out-parameters.

[1] http://seclists.org/bugtraq/1997/Jan/11

5.1.9. ERR07-C. Prefer functions that support error checking over equivalent functions that don't

Some C standard library functions like atoi give no error indication, and they even have undefined behavior in case of error! They should not be used in any program, especially not security-critical one. Even suggested replacement, strtol, is not perfect. Its flexibility (e.g. returning pointer to first invalid character) makes it harder to check for all error conditions. hardened-shadow uses hardened_shadow_strtonum, inspired by OpenBSD's strtonum(3).

5.2. asprintf

String handling is not the strongest part of the C language, and may even become a security risk. asprintf is a GNU extension, which works like sprintf, but allocates the target buffer itself. It is a very useful feature, because it relieves the programmer from burden of calculating the necessary buffer size, which is error-prone. It also results in shorter, easier to read code that is more secure and robust.

For example, the following procedure is short and readable. Manually calculating buffer size would be more difficult, error-prone, and result in more code:

```
bool hardened_shadow_asprintf_aging(char **result, const
   struct spwd *spwd) {
  if (asprintf(result,
               "%ld:%ld:%ld:%ld:%ld:\n",
               spwd->sp_min,
               spwd->sp_max,
               spwd->sp_warn,
               spwd->sp_inact,
               spwd->sp_expire) == -1) {
    return false;
  }
  return true;
}
```

5.3. err.h

There is a very nice set of error reporting functions originating from OpenBSD. The functions are defined in err.h (this is also a non-standard extension).

Below is an example from login(1) program:

```
  char **pam_env = pam_getenvlist(pam_handle);
  if (!pam_env)
    errx(EXIT_FAILURE, "pam_getenvlist␣returned␣NULL");

  struct environment_options environment_options = {
    .pam_environment = pam_env,
    .preserve_environment = preserve_environment,
    .login_shell = true,
    .target_username = pam_user,
```

```
    .target_homedir = target_homedir,
    .target_shell = target_shell,
  };
  if (!hardened_shadow_prepare_environment(&environment_options
     ))
    errx(EXIT_FAILURE, "hardened_shadow_prepare_environment");

  if (setuid(target_uid) != 0)
    err(EXIT_FAILURE, "setuid");
```

5.4. stdbool

bool type is arguably cleaner than int for booleans. true and false look better than 1 and 0, and including stdbool.h is cleaner than custom macros.

5.5. handling EINTR

It is often forgotten that many UNIX calls can fail with EINTR error, i.e. system call interrupted by a signal. Software not prepared for this may be prone to random failures, e.g. SIGCHLD at just the wrong time. The easiest way to handle EINTR is to wrap the calls with TEMP_FAILURE_RETRY from unistd.h (it is a GNU extension).

Examples:

```
/* write(2) wrapper that handles partial writes and EINTR
   correctly. */
ssize_t hardened_shadow_write(int fd, const char *data, size_t
   size) {
  size_t total = 0;
  for (ssize_t partial = 0; total < size; total += partial) {
    partial = TEMP_FAILURE_RETRY(write(fd, data + total, size -
        total));
    if (partial < 0)
      return partial;
  }
  return total;
}

/* read(2) wrapper that handles partial reads and EINTR
   correctly. */
ssize_t hardened_shadow_read(int fd, char *data, size_t size) {
  size_t total = 0;
  for (ssize_t partial = 0; total < size; total += partial) {
    partial = TEMP_FAILURE_RETRY(read(fd, data + total, size -
       total));
    if (partial < 0)
      return partial;
    if (partial == 0)
      break;
```

```
  }
  return total;
}
```

5.6. implicit library cleanup

hardened-shadow's NSSwitch module is implicitly loaded by glibc into programs running in user space. Because these programs are not aware of the specific module (that is the whole idea behind NSSwitch), neither the initialization nor cleanup can be explicit, and has to be implicit instead. Fortunately, gcc provides a useful attribute for that:

```
static void __attribute__((destructor))
   hardened_shadow_fd_cleanup(void) {
  if (internal_hardened_shadow_fd != -1)
    TEMP_FAILURE_RETRY(close(internal_hardened_shadow_fd));
}
```

5.7. fmemopen

glibc provides convenient functions like fgetgrent(3), but they operate on file handles, not strings. hardened-shadow sometimes needs an equivalent call but working on in-memory strings. Here is how this is solved:

```
/* fgetgrent wrapper that takes a memory buffer instead of a
   file. */
struct group *hardened_shadow_sgetgrent(char *buf) {
  FILE *stream = fmemopen(buf, strlen(buf), "r");
  if (!stream)
    return NULL;
  struct group *result = fgetgrent(stream);
  TEMP_FAILURE_RETRY(fclose(stream));
  return result;
}
```

Chapter 6

Open Source Project

Since the inception of hardened-shadow project I wanted to make it available to the world as an Open Source project. The most important reason was to get feedback from experienced programmers, system administrators and distribution packagers that are interested in making UNIX systems more secure. Obviously getting as many eyes as possible to look at a given codebase is going to produce more comments than if only two or three people took a look. That is not a guarantee of security or corrctness however, only one way to assess them.

6.1. Choosing the name

Name of the project is surprisingly important. It should be easy to search for on the internet, and easy to remember. Also, it should not be too close to existing project names, to avoid possible confusion or even trademark infringement.

I considered vsshadow first, as a reference to FTP daemon vsftpd (very secure FTP daemon), but decided not to use it because the meaning of "vs" is not obvious (it even might suggest "versus" instead of "very secure").

hardened-shadow is pretty obvious when one knows what password shadowing is. The first part of the name suggests some kind of stronger security. While a search for that term also returns unrelated results, the project page is the first one.

6.2. Licensing

Initially I used a 3-clause BSD license for code written entirely by me. For all other code I preserved original license headers and only added my copyright notice. After a comment[1] by Solar Designer I switched to 2-clause BSD for simplicity.

The codebase is small (less than 10 KLOC [11]), and is already based on BSD-licensed files, so BSD was a natural choice. Due to the small size GPL was not attractive anyway.

I also made it clear on the project page that the project is free and open source. It is important because not everyone is familiar with different licenses, and it is not convenient to search inside packages to look for license headers or license files.

[1] http://openwall.com/lists/owl-dev/2012/03/14/2

6.3. Hosting

I chose code.google.com as the hosting provider for the project. I already had excellent experience with Google Code after working on the Chromium project hosted by Google (Chromium is the Open Source project behind Google Chrome). code.google.com is fast, simple, and "just works".

The project mailing lists are hosted on groups.google.com, for a similar reason. There are multiple groups: hardened-shadow-dev for technical development discussions, and hardened-shadow-users for any user questions, discussions and support. There are also two potentially high-traffic lists: hardened-shadow-commits (notifications about pushes to the official code repository) and hardened-shadow-bugs (notifications about bug racker activity).

I chose git for version control, because of its distributed nature (it would be a reasonable use case to fork the official repository in order to send substantial code contributions to the project) and because it is fairly well known. Note that for such a small project this choice does not matter that much. It is relatively easy to change, which is not the case for huge projects.

6.4. Testing

So far no bug reports have been filed in the tracker, although some people have definitely downloaded and at least compiled the code.

The author has been using hardened-shadow on his Linux system since November 2011. Some bugs in the early implementation led to various lockouts, but daily usage helped to quickly identify and fix them.

6.5. Releases

Three releases have been made: 0.9, 0.9.1 and 0.9.2. Odd minor version number means an unstable version (this is a common convention).

Each release is tagged in the version control system. Both the tag and the release tarball are signed using my GPG key. Note that this is not an "extra" security measure and can be reasonably expected. shadow-utils releases are also cryptographically signed.

It is worth noting that some people have downloaded hardened-shadow signature files, presumably to verify them. That is a good sign, and also indicates that they are expected and useful.

6.6. Packaging

I packaged hardened-shadow for Gentoo Linux, and continue to maintain it (I am Gentoo developer since 2009). This was not a trivial operation, because up to this point the distribution was using shadow-utils as the only implementation. First I had to create a virtual package (virtual/shadow) and modify the profiles (metadata which e.g. lists the packages that must be installed on every system) accordingly.

Then it turned out Gentoo has custom PAM changes that were part of sys-apps/shadow package. To avoid duplication, I had to do a new release of pambase, which is a collection of Gentoo-made pam.d files. Version bumps of shadow packages followed, and a breakage[2]

[2] https://bugs.gentoo.org/show_bug.cgi?id=412721

causing system lockout has been reported. However, it turned out the user failed to update configuration files in /etc, and for some reason the reminder to do that has not been displayed. In the end the issue has been categorized as user mistake.

Another bug[3] has been filed about file collision issue on Gentoo/FreeBSD. It turns out that sys-freebsd/freebsd-ubin package also provides /etc/pam.d/login, passwd, and su files that my change added to pambase. To fix this another pambase version will be released, with a special case for FreeBSD added.

[3] https://bugs.gentoo.org/show_bug.cgi?id=413077

Chapter 7

Summary

In this chapter the main achievements of the hardened-shadow project are summarized.

7.1. Goals achieved

I have successfully created an Open Source package of UNIX password utilities, which has reduced attack surface and has been carefully implemented to achieve excellent code readability and follow least privilege principle and state-of-the-art secure coding guidelines to increase security.

This alternative implementation has been provided for free on the internet, and is available as part of Gentoo Linux distribution. It should work with other Linux distributions with at most minimal changes. The project received extensive coverage[1] on LWN.net (Linux Weekly News), which is the premier news source for the free software community.

7.2. Reception

7.2.1. Openwall

In an e-mail thread[2] on owl-dev mailing list, Solar Designer, a highly respected security specialist from Russia (his real name is Alexander Peslyak), has praised hardened-shadow in the following words:

```
On Wed, Mar 14, 2012 at 12:46:12PM +0100, Pawel Hajdan, Jr. wrote:
> I'd like to announce my little project I've published recently:
> hardened-shadow.

Thank you for bringing this to owl-dev.  You may also want to announce
it on oss-security (maybe a bit later).

It's an impressive amount of work you did.
```

[1] http://lwn.net/Articles/487620/
[2] http://openwall.com/lists/owl-dev/2012/03/14/2

In a later e-mail, he also confirms my opinion that password-protected groups are not really a good idea (they are not implemented in hardened-shadow):

```
On Thu, Mar 15, 2012 at 05:03:09PM +0100, Pawel Hajdan, Jr. wrote:
> On Wed, Mar 14, 2012 at 23:53, Solar Designer <solar@...nwall.com> wrote:
> > FWIW, I noticed that you also excluded gpasswd - you could want to
> > document that in your list of missing features.
>
> Right, that was also on purpose - I think nowadays password-protected
> groups are not really used, and they increase complexity of the tools.

I agree.
```

hardened-shadow may become a part of Openwall Linux distribution, but before that eventually happens many details would have to be figured out. It is somewhat of an accomplishment to write code that is considered for inclusion in Openwall:

```
We haven't decided on our possible use of it yet, but we'll need to
consider moving to it as an alternative to upgrading the shadow suite in
Owl (and forward-porting our patches).  Moreover, we'll also need to
decide on staying with SimplePAMApps + patches vs. moving to your
implementations of these programs.  As to PAM and NSS modules, I think
we'll just stay with ours (so we'll have almost no incentive to audit
yours even if we do use other parts of hardened-shadow).
```

Note how hardened-shadow provides a viable alternative to the existing shadow suites with custom patches. The hard work put into this implementation has been recognized.

7.2.2. LWN

In March 2012 an article[3] appeared on LWN about shadow hardening. It is a notable achievement to be mentioned by LWN, and the article itself is pretty long, gives a lot of details and context. Some notable paragraphs are quoted below:

> Nevertheless, Peslyak said that Openwall would consider migrating to hardened-shadow's version of the shadow utilities (although not the PAM and NSS modules). The reason he gave was that Openwall currently maintains its patch set against a suite of tools that come from a blend of two different sources: the canonical shadow utilities, and a PAM-based implementation called SimplePAMApps. SimplePAMApps, however, is no longer being actively maintained. Owl and a few other distributions curate their own packages of it, but in the long run, synchronizing with an actively developed tool set is probably less work.
>
> Back in 2010, Hajdan submitted patches to the shadow utilities to enable optional support for tcb, a change that appears to have landed in the 4.1.5 release from February 2012. Nevertheless, tcb is still an essentially Owl-only tool. There does not seem to be much interest in packaging it among the large, "mainstream" distributions; in 2006 the idea was floated on the fedora-devel list where it was met with skepticism.

[3] http://lwn.net/Articles/487620/

> Some of the criticism from 2006 is questionable (for example, the concern that users could fill up the /etc filesystem by dumping files into their /tcb/ directories — the group permissions should prevent that), but others, such as requiring new methods for auditing password changes, may be more valid. Hardened-shadow does not alleviate all of those concerns, but by virtue of not requiring a patched Glibc, it at least stands a better chance of being packaged by other distributions.

Note that hardened-shadow is available in Gentoo Linux's package repository, and it is a step forward from tcb that requires too many custom patches.

7.3. Further development

There are many areas which can be developed further in hardened-shadow. Thanks to its Open Source nature, the community is welcome to participate in that process.

In fact, hardened-shadow may be useful in Operating Systems classes or labs, with its small, readable codebase and many enhancements still left to be done. It can also be used during Computer and Network Security and similar classes.

7.3.1. Comprehensive security audit

It is not enough to just be careful and follow best practices during implementation. It is very easy to introduce bugs into software, and many of them are security vulnerabilities. hardened-shadow needs multiple security experts looking at the code, and trying to break it in various ways. Also, usage in more real-world scenarios (e.g. hosting a shell server, where users are not fully trusted) is likely to reveal some security holes.

7.3.2. Testing on SELinux-enabled systems

While hardened-shadow has been designed to rely only on traditional UNIX permissions for security, there are systems in use with strict SELinux policies that are likely not to work out-of-the-box with hardened-shadow. One obvious reason would be using /etc/hardened-shadow directory tree instead of the /etc/shadow file. Because of the least-privilege philosophy adopted by SELinux, access to resources not explicitly allowed would be denied, even to system utilities.

7.3.3. Support for traditional /etc/shadow

To make transition even easier for existing systems, support for /etc/shadow may be added. Doing so would require running some utilities with further elevated privileges. However, in real world easy upgrade paths are essential for new software adoption. All security benefits of hardened-shadow are beter utilized when more system administrators use it. Providing a safe "middle step" that does not require conversion of shadow file is also useful for Linux distributions, which also usually try to avoid breaking changes.

7.3.4. Plugin-based architecture

Instead of implementing just a compatibility-mode for /etc/shadow, a whole plugin interface could be created, so that the same command-line interface could work with various backends.

hardened-shadow benefits a lot from PAM and NSSwitch, which bring such plugin-based architecture for authentication and user account management respectively.

7.3.5. Usage in embedded systems

hardened-shadow has been deliberately kept small and simple. While it would still need more optimizations focused on code size and dependencies, it might have an advantage by having most of rarely used features already removed from the codebase.

7.3.6. Porting to other operating systems

Currently the project has only beed developed and tested on Linux, and is likely relying on some Linux-specific interfaces and behaviors. Porting to other systems like BSD variants should not be too hard however, as I have generally used POSIX and other standard interfaces where possible.

7.4. Conclusion

hardened-shadow is a successful project that received praise from respected security specialists like Solar Designer, is available in Gentoo Linux's package repository, and is considered for inclusion in other distributions. It addresses concerns raised about the need to custom-patch other packages when using Openwall's tcb, and provides significantly stronger security by reducing the attack surface and careful implementation techniques. There are plenty of opportunities for further development, and the project can be useful both for industry and academia.

Appendix A

Attached CD

Attached ISO-9660 CD contains git repository used to develop hardened-shadow and electronic version of the thesis.

A.1. Directory structure

- 1000-MGR-89012007292.pdf — electronic version of this thesis,
- hardened-shadow,
  - README
  - Makefile.am — automake file,
  - configure.ac — autoconf file,
  - etc
    - hardened-shadow.conf — main configuration file,
    - securetty — list of terminals on which root is allowed to login,
    - default — default settings,
      - useradd — defaults for useradd,
    - pam.d — PAM configuration files,
      - login — PAM settings for login,
      - passwd — PAM settings for passwd,
      - su — PAM settings for su,
  - common
    - config.c — configuration routines (hardened-shadow.conf),
    - dir.c — walking directory tree: copying, deleting directories with contents,
    - file.c — working on files: opening, reading, writing, replacing,
    - hardened-shadow.h — header containing declaration of all functions in this directory,
    - misc.c — miscellanous functions,
    - parse.c — parsing strings, e.g. string to number, string to date conversions,
    - print.c — output formatting, e.g. date to string conversion,
    - pwck.c — integrity checking of the password files,
    - safeint.c — overflow/wraparound checking for integer operations,

    - syslog.c — logging,
  - m4
    - attributes.m4[1] — m4 macros for detecting compiler flags support,
    - xml.m4[2] — m4 macros for working with xml files,
  - man[3] — man pages,
  - nsswitch
    - hardened_shadow.c — NSSwitch module,
    - hardened_shadow.map — symbol visibility settings for the linker,
  - pam[4],
    - pam.map — symbol visibility settings for the linker,
    - pam_hardened_shadow_acct.c — implementation of PAM account hook,
    - pam_hardened_shadow_auth.c — implementation of PAM authentication hook,
    - pam_hardened_shadow_passwd.c — implementation of PAM password hook,
    - pam_hardened_shadow_sess.c — implementation of PAM session hook,
    - support.c — common routines,
    - support.h — declarations of functions implemented in support.c,
  - progs — programs, one file per program.

A.2. How to build from git

A.2.1. Prerequisites

- autoconf,
- automake,
- libtool,
- libxslt,
- make,
- Linux-PAM,
- DocBook DTD 4.5 and XSL Stylesheet.

A.2.2. Instructions

```
libtoolize
autoreconf -i
./configure --enable-generate-man
make
```

[1] taken from xine project
[2] taken from shadow-utils project
[3] based on shadow-utils man pages
[4] based on Linux-PAM code

A.3. How to install

1. Make sure wheel and hardened-shadow groups exist.
2. Run pwck and fix as many errors as possible.
3. Run make install (as root).
4. Run pwconv (as root).
5. Replace "shadow: compat" line in /etc/nsswitch.conf with "shadow: hardened_shadow".
6. Replace "pam_unix.so" with "pam_hardened_shadow.so" in /etc/pam.d.
7. Make sure pam_hardened_shadow.so for passwd has parameter "prefix=6".
8. Add /bin/shell_proxy to /etc/shells.

Appendix B

Full text of used software license (2-clause BSD)

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE

Appendix C

Safe integer operation routines

In most C projects integer checks are performed "manually", inline in the code. This is an error-prone practice, and often even these checks are skipped. hardened-shadow contains an easy to use set of routines for checking integer operation, developed based on the C secure coding standard [9]. The routines are important part of the thesis, and so have been included here for review.

C.1. Unsigned operations

```
bool hardened_shadow_ucast_ok(intmax_t a, uintmax_t max) {
  return a > 0 && (uintmax_t)a < max;
}

bool hardened_shadow_uadd_ok(uintmax_t a, uintmax_t b,
   uintmax_t max) {
  if (a > max || b > max)
    return false;
  return max - a >= b;
}

bool hardened_shadow_usub_ok(uintmax_t a, uintmax_t b,
   uintmax_t max UNUSED) {
  return a >= b;
}

bool hardened_shadow_umul_ok(uintmax_t a, uintmax_t b,
   uintmax_t max) {
  if (a > max || b > max)
    return false;
  return a <= max / b;
}
```

C.2. Signed integer operations

```
bool hardened_shadow_scast_ok(uintmax_t a, intmax_t max) {
  return a < (uintmax_t)max;
```

```
}

bool hardened_shadow_sadd_ok(intmax_t a,
                             intmax_t b,
                             intmax_t min,
                             intmax_t max) {
  if (a > max || b > max || a < min || b < min)
    return false;
  if (b > 0 && a > (max - b))
    return false;
  if (b < 0 && a < (min - b))
    return false;
  return true;
}

bool hardened_shadow_ssub_ok(intmax_t a,
                             intmax_t b,
                             intmax_t min,
                             intmax_t max) {
  if (a > max || b > max || a < min || b < min)
    return false;
  if (b > 0 && a < (min + b))
    return false;
  if (b < 0 && a > (max + b))
    return false;
  return true;
}

bool hardened_shadow_smul_ok(intmax_t a,
                             intmax_t b,
                             intmax_t min,
                             intmax_t max) {
  if (a > max || b > max || a < min || b < min)
    return false;
  if (a > 0) {
    if (b > 0 && a > (max / b))
      return false;
    if (b < 0 && b < (min / a))
      return false;
  } else {
    if (b > 0 && a < (min / b))
      return false;
    if (b < 0 && a != 0 && b < (max / a))
      return false;
  }
  return true;
}

bool hardened_shadow_sdiv_ok(intmax_t a,
```

```
                                intmax_t b,
                                intmax_t min,
                                intmax_t max) {
  if (a > max || b > max || a < min || b < min)
    return false;
  if (b == 0)
    return false;
  if (a == min && b == -1)
    return false;
  return true;
}
```

Appendix D

Environment sanitization routine

Environment sanitization is another practice recommended by C secure coding standard [9]. This appendix contains code of the environment sanitization routine used by hardened-shadow, to show that it is harder than it may seem. The code is pretty short and straightforward, but that is not 2-3 lines either. It is important to understand meaning of many environment variables on UNIX, such as IFS (internal field separator for the shell; it changes the way shell interprets commands and can be abused to make shell execute unexpected commands).

```
static const char *kLoginPreservedEnv[] = {
  "TERM",
  "COLORTERM",
  "DISPLAY",
  "XAUTHORITY",
};

bool hardened_shadow_prepare_environment(
    const struct environment_options *options) {
  if (!options->preserve_environment) {
    char* preserved_variables[HARDENED_SHADOW_ARRAYSIZE(
       kLoginPreservedEnv)];
    if (options->login_shell) {
      for (size_t i = 0;
           i < HARDENED_SHADOW_ARRAYSIZE(kLoginPreservedEnv);
           i++) {
        if (getenv(kLoginPreservedEnv[i])) {
          preserved_variables[i] = strdup(getenv(
             kLoginPreservedEnv[i]));
          if (!preserved_variables[i]) {
            hardened_shadow_syslog(LOG_ERR, "memory␣allocation␣
               failure");
            return false;
          }
        } else {
          preserved_variables[i] = NULL;
        }
      }
```

```
}
if (clearenv() != 0) {
  hardened_shadow_syslog(LOG_ERR, "clearenv failed");
  return false;
}
if (options->login_shell) {
  for (size_t i = 0;
       i < HARDENED_SHADOW_ARRAYSIZE(kLoginPreservedEnv);
       i++) {
    if (!preserved_variables[i])
      continue;
    if (setenv(kLoginPreservedEnv[i], preserved_variables[i
       ], 1) != 0) {
      hardened_shadow_syslog(LOG_ERR, "setenv failed");
      return false;
    }
    free(preserved_variables[i]);
  }

  /* Figure out an existing homedir, and set $HOME
     accordingly. */
  if (chdir(options->target_homedir) == 0) {
    if (setenv("HOME", options->target_homedir, 1) != 0) {
      hardened_shadow_syslog(LOG_ERR, "setenv failed");
      return false;
    }
  } else if (chdir("/") == 0) {
    if (setenv("HOME", "/", 1) != 0) {
      hardened_shadow_syslog(LOG_ERR, "setenv failed");
      return false;
    }
    puts("No directory, logging in with HOME=/");
  } else {
    hardened_shadow_syslog(LOG_ERR,
                           "unable to cd to `%s' for user
                              `%s'",
                           options->target_homedir,
                           options->target_username);
    return false;
  }

  if (setenv("PATH", "/bin:/usr/bin", 1) != 0) {
    hardened_shadow_syslog(LOG_ERR, "setenv failed");
    return false;
  }
} else {
  /* Not a login shell. */

  if (setenv("HOME", options->target_homedir, 1) != 0) {
```

```
        hardened_shadow_syslog(LOG_ERR, "setenv␣failed");
        return false;
      }
    }

    if (setenv("SHELL", options->target_shell, 1) != 0) {
      hardened_shadow_syslog(LOG_ERR, "setenv␣failed");
      return false;
    }
    if (setenv("USER", options->target_username, 1) != 0) {
      hardened_shadow_syslog(LOG_ERR, "setenv␣failed");
      return false;
    }
    if (setenv("LOGNAME", options->target_username, 1) != 0) {
      hardened_shadow_syslog(LOG_ERR, "setenv␣failed");
      return false;
    }

    char **env_iter = options->pam_environment;
    while (*env_iter) {
      char *pos = strchr(*env_iter, '=');
      if (!pos) {
        hardened_shadow_syslog(LOG_ERR, "pam_environment␣is␣
           invalid");
        return false;
      }
      *pos = '\0';
      if (setenv(*env_iter, pos + 1, 1) != 0) {
        hardened_shadow_syslog(LOG_ERR, "setenv_failed");
        return false;
      }
      env_iter++;
    }
  }

  if (setenv("IFS", "␣\t\n", 1) != 0) {
    hardened_shadow_syslog(LOG_ERR, "setenv␣failed");
    return false;
  }

  return true;
}
```

Bibliography

[1] Steven Alexander, *Password Protection for Modern Operating Systems*, USENIX ;login **29** (2004), no. 3.

[2] Hao Chen, David Wagner, and Drew Dean, *Setuid Demystified*, Proceedings of the 11th USENIX Security Symposium (San Francisco, CA), August 5–9, 2002, pp. 171–190.

[3] Karl Fogel, *Producing Open Source Software: How to Run a Successful Free Software Project*, O'Reilly Media, Inc., 2005.

[4] Andrew Hunt and David Thomas, *The pragmatic programmer: from journeyman to master*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[5] ISO, *ISO C Standard 1999*, Tech. report, 1999, ISO/IEC 9899:1999 draft.

[6] Michael Kerrisk, *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*, 1st ed., No Starch Press, San Francisco, CA, USA, 2010.

[7] Peter Miller, *Recursive Make Considered Harmful*, AUUGN Journal of AUUG Inc. **19** (1998), no. 1, 14–25.

[8] Niels Provos and David Mazières, *A Future-Adaptable Password Scheme*, USENIX Annual Technical Conference, FREENIX Track, 1999, pp. 81–91.

[9] Robert C. Seacord, *The CERT C Secure Coding Standard*, 1st ed., Addison-Wesley Professional, 2008.

[10] Dan Tsafrir, Dilma Da Silva, and David Wagner, *The murky issue of changing process identity: revising "setuid demystified"*, USENIX ;login **33** (2008), no. 3, 55–66.

[11] David Wheeler, *SLOCcount*, 2009.