Transforming Code to Drop Dead Privileges

Xiaoyu Hu
BitFusion.io Inc.

Jie Zhou
Department of Computer Science
University of Rochester

Spyridoula Gravani
Department of Computer Science
University of Rochester

John Criswell
Department of Computer Science
University of Rochester

***Abstract*—To help programmers write programs that follow Saltzer and Schroeder's Principle of Least Privilege, modern operating systems divide the power of the administrative user into separate privileges which applications can enable on demand and permanently discard when no longer needed. However, using such privileges requires tedious temporal reasoning of program behavior.**

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

I. Introduction

Saltzer and Schroeder's Principle of Least Privilege [1] dictates that software should only have the privileges that it needs to operate correctly. To help programmers adhere to this principle, modern operating systems like Linux [2], Windows [3], and Solaris [4] divide the power of the administrative user into separate privileges. In these operating systems, a process has an effective set of privileges that the operating system kernel consults during access control checks and a maximum privilege set which the process can reduce but never increase [2], [3], [4]. This design allows processes to *raise* a privilege from the maximum set into the effective set so that it can selectively use privileges when needed. It also allows processes to permanently *remove* a privilege from both the effective and maximum sets so that the process can never use the privilege again.

For example, the Linux kernel provides separate privileges for binding to privileged TCP ports and overriding access controls when opening files for reading [5]. A network server that only needs to bind to privileged TCP ports need not have the ability to read any file on the system. Furthermore, once the server has bound all of its listening sockets to privileged TCP ports, it can permanently remove its ability to bind to privileged ports so that an attacker that exploits the server cannot use the privilege.

Permanently removing privileges is especially important for programs written in type-unsafe programming languages. If such a program has a memory safety error, an attacker could use a return-to-libc [6] or return-oriented programming [7] attack to raise privileges that have been temporarily disabled (or *lowered* [8]) in the effective set but that still remain in the maximum set. Removing privileges from both the effective and maximum sets prevents such privilege escalation attacks.

Programmers face a challenge when writing programs that use a minimum of privilege. Since a process can misuse any privilege in its maximum privilege set (even if the privilege is lowered), programs need to permanently remove privileges as soon as they can. Determining when a program can safely remove a privilege requires reasoning about the program's use of system calls across function boundaries and compilation units. Programmers need an automated tool that locates program points at which privileges can be disabled permanently.

In this paper, we present a tool named AutoPriv that aids programmers in solving this challenge. AutoPriv is a compiler that takes, as input, a program that raises and lowers operating system privileges and outputs a transformed program that keeps those privileges in the maximum privilege set only as long as they may be needed. AutoPriv utilizes inter-procedural compiler analysis to determine where privileges are used and at what program points they can be safely removed from the maximum privilege set.

Our contributions are as follows:

- We develop an inter-procedural analysis that determines when privileges can safely be removed from a program without altering its behavior.
- We develop an inter-procedural compiler transform that modifies a program to permanently remove operating system privileges granted to it when the privileges are no longer needed.
- We tested our prototype on five open-source programs that typically run as the root user. We show that transforming programs to use the least operating system privilege increases optimization time by 19% on average across our benchmarks but practically induces no overhead on compiled programs.

The rest of this paper is organized as follows: Section II discusses the Linux privilege system. Section III presents our attack model. Section IV discusses our privilege analysis and transformation algorithms. Section V describes our prototype's implementation, and Section VI describes our prototype's performance. Section VII describes related work, and Section VIII describes future work and concludes.

II. Linux Privileges

Traditional Unix discretionary access controls allow a process running with an effective user ID of zero, i.e., the root user, to bypass all access control checks [9], [2]. Linux

Fig. 1: AutoPriv Architecture

separates the powers of the root user into separate privileges (which Linux calls *capabilities*) [5]. Each privilege overrides a subset of the discretionary access control rules; for example, the `CAP_CHOWN` capability allows a process to change a file's owner while `CAP_NET_BIND_SERVICE` allows a process to bind to a privileged TCP/UDP port [5]. We will use *privilege* and *capability* interchangeably due to the Linux nomenclature.

All Linux processes have three capability sets: the *effective* set, the *permitted* set, and the *inheritable* set [5]. The effective set is the set of capabilities which the kernel checks when determining if a process is allowed to override an access control; the permitted set is a superset of both the inheritable and effective sets [5]. The inheritable set is the set of capabilities that the process retains if it executes a new program and the attributes on the executable file in the file system permit the process to retain those capabilities [5]; it provides a way for a subset of capabilities to "carry over" an `exec()` system call.

Linux processes can modify their privilege sets using the `cap_set_flag()` system call [5]. The inheritable and effective sets can be changed to a subset of the permitted set, and a process can always reduce its permitted set to a subset of the current permitted set; only processes with the appropriate capability can add new capabilities into their permitted set [5].

A program designed to follow the Principle of Least Privilege will only enable capabilities in the effective set for system calls in which it needs to override an access control rule and disable the capability in its effective set when it does not need to use it; following the terminology from the Compartmented Mode Workstation [8], we call this *raising* and *lowering* a privilege, respectively. Such a program will also remove capabilities from the effective and permitted sets when the program will no longer use those capabilities. We call this *removing* the privilege.

Refactoring a program to raise and lower capabilities around system calls is straight-forward as the set of Linux capabilities [2] is small and easy to understand. Removing capabilities requires temporal and inter-procedural reasoning about a program's behavior; if a process may use the capability later in its execution, it must keep the capability in the permitted set until the very last use.

III. Attack Model

Our attack model assumes that the processor and operating system kernel are implemented correctly and are part of the Trusted Computing Base (TCB). However, an attacker can corrupt an application's control-flow and data-flow to alter the inputs to systems calls and to execute system calls in an order not possible in the original program [10], [11], [7]. Since processes use system calls to raise privileges into the effective privilege set, our attack model permits attackers to raise and use any privilege that is present in the process's permitted privilege set. To prevent attacks from raising and abusing its privileges, a program should remove privileges from its permitted set as soon as possible.

IV. Design

Figure 1 shows AutoPriv's overall design. Given a program that has been refactored to raise and lower privileges before and after system calls and library calls that use privileges, AutoPriv will analyze the privilege use within the program, compute the sets of privileges that are live on entry and exit to each basic block i.e., the *live privilege sets* in Figure 1, add code to remove privileges when they are no longer needed, and generate a final executable.

AutoPriv makes a few assumptions about how programs use privileges. First, AutoPriv assumes that each program has a known set of privileges that it needs to execute; privileges are not inherited across programs via the `exec()` family of system calls. In Linux, this means that processes have an empty inheritable set. Second, AutoPriv assumes that a programmer has added code to raise and lower privileges (known as *privilege bracketing* [8]). Third, AutoPriv assumes that external library code does not raise and lower privileges. Rather, the programmer brackets calls to external library functions needing privileges with calls to raise and lower privileges. In our experience, libraries such as the C library do not raise and lower privileges.

A. Privilege Primitives

To provide a simpler interface for manipulating process capability sets, we created a library to wrap the Linux `cap_get_flag()` and `cap_set_flag()` system calls. This library provides a set of privilege manipulation primitives that we use throughout our work. We based these primitives after those in the PitBull Foundation system [12] (which, in turn, are based on those provided by Berger et al. [8]). Our privilege primitives are as follows:

1) **`priv_raise` (int cap_num, int capability, ...):** The `priv_raise` primitive takes a list of capabilities and adds them to the process's effective capability set. The permitted set remains unchanged.
2) **`priv_lower` (int cap_num, int capability, ...):** The `priv_lower` primitive takes a list of capabilities and removes them from the process's effective capability set. The permitted set remains unchanged.
3) **`priv_lowerall` ():** The `priv_lowerall` primitive disables all capabilities in the process's effective capability set. The permitted set remains unchanged.
4) **`priv_remove` (int cap_num, int capability, ...):** The `priv_remove` primitive removes all the specified

Algorithm 1 Local Privilege Analysis Algorithm

```
1:  CAPUse: Map from basic blocks to used capability sets
2:
3:  function LocalPrivilegeAnalysis(CAPUse)
4:      for all functions F in program do
5:          for all basic blocks BB in function F do
6:              for all calls C to priv_raise in BB do
7:                  PrivSet ← privileges raised in C
8:                  CAPUse[BB] ← CAPUse[BB] ∪ PrivSet
9:              end for
10:         end for
11:     end for
12:     return
13: end function
```

capabilities from *both* the effective and permitted capability sets. Once a capability is removed, the process can never possess it again.

To refactor a program, the programmer adds calls to `priv_raise` and `priv_lower` around system calls and library calls that need to use capabilities. AutoPriv adds calls to `priv_remove` to permanently disable capabilities when they are no longer needed; it also adds a call to `priv_lowerall` within the program's `main()` function to ensure that all capabilities are lowered upon program start.

Adding calls to `priv_remove` burdens programmers. A call to `priv_remove` can safely occur only at points in the program in which the removed privileges will not be used again. Determining such points requires global reasoning about the program; local examination of a function or method alone does not suffice for determining where to insert `priv_remove` calls. Additionally, small changes to a program can drastically change the points at which `priv_remove` can be safely called (because an additional function call can increase the program points at which a privilege is still needed).

To ease programmer burden, we have developed a compiler analysis and transformation that automatically inserts calls to `priv_remove`.

B. Live Privilege Analysis

To determine at which points in a program to add calls to `priv_remove`, AutoPriv must determine which privileges can still be used at each point in the program. We define the *live capabilities* at a program point p to be the capabilities that may still be used (via a `priv_raise`) along some path in the program. Our live capabilities definition is analogous to the definition of *live variables* in live variable analysis [13].

We have developed an inter-procedural, flow-insensitive, context-insensitive live capability (i.e. live privilege) analysis based on the standard iterative data-flow analysis framework developed by Kam and Ullman [13]; this analysis computes the live capability sets at the beginning and end of each basic block. Our compiler then uses this information to locate points in a program at which the live capability sets change in order to locate where it can safely insert calls to `priv_remove`. To simplify the presentation of our algorithms, we assume that each basic block contains only a single instruction. Our prototype implementation described in Section V relaxes this requirement.

Algorithm 2 Global Privilege Analysis Algorithm

```
1:  CAPUse: Map from basic blocks to used capability sets
2:  funcOut: Map from functions to used capability sets
3:
4:  function GlobalPrivAnalysis(CAPUse, funcOut)
5:      for each function F in program do
6:          funcOut[F] ← ∅
7:          for each basic block BB in F do
8:              funcOut[F] ← funcOut[F] ∪ CAPUse[BB]
9:          end for
10:     end for
11:
12:     repeat
13:         changed ← False
14:         for each function F in program do
15:             for each child CF of F in the call graph do
16:                 if funcOut[CF] ⊈ funcOut[F] then
17:                     funcOut[F] ← funcOut[F] ∪ funcOut[CF]
18:                     changed ← True
19:                 end if
20:             end for
21:         end for
22:     until changed = False
23:     return
24: end function
```

To eliminate imprecision due to infeasible paths, our global live privilege analysis uses a *global privilege analysis* algorithm to compute function summaries of privilege use for each function. Specifically, it computes a summary of the privileges used by a function and the functions it can call (either directly or transitively). The global privilege analysis algorithm, in turn, uses a helper analysis called the *local privilege analysis* algorithm to find which privileges are raised in each basic block within the program.

1) Local Privilege Analysis: The local privilege analysis algorithm determines which privileges are raised by each basic block. Described by Algorithm 1, the analysis locates all calls to `priv_raise` and examines the privileges enabled at each call to `priv_raise`. The analysis then records the mapping between basic blocks that call `priv_raise` and the privileges enabled by those calls.

Algorithm 1 provides a straightforward, compiler-agnostic algorithm for the analysis. Compilers that maintain explicit definition-use chains in their intermediate representations (such as LLVM [14]) can optimize Algorithm 1 by finding all uses of the `priv_raise` function that are call instructions. While both algorithms are linear-time (with respect to program size), the latter can be more efficient.

2) Global Privilege Analysis: Once the Local Privilege Analysis pass has computed which privileges are used within each basic block, the Global Privilege Analysis pass can compute the privileges used by a function and all of its callees (i.e., which privileges could ever be used when a function is called).

Algorithm 2 provides pseudo-code for the Global Privilege Analysis pass. This algorithm creates a map between *functions* and the capabilities that they use (either directly or transitively). Algorithm 2 first aggregates the capabilities used in

Algorithm 3 Global Live Privilege Analysis Algorithm

```
1:  BBIn: Map from basic blocks to capability set live on entry to block
2:  BBOut: Map from basic blocks to capability set live on exit from block
3:  funcOut: Map from functions to used capability sets
4:
5:  function GLOBALLIVEPRIVILEGEANALYSIS(funcOut, BBIn, BBOut)
6:      repeat
7:          changed ← False
8:          for all functions func in program do
9:              repeat
10:                 bbChanged ← False
11:                 for all Basic Blocks BB in func do
12:                     oldIn ← BBIn[BB]
13:                     oldOut ← BBOut[BB]
14:
15:                     // Propagate live privileges from all successor blocks
16:                     for all successors S of BB do
17:                         BBOut[BB] ← BBOut[BB] ∪ BBIn[S]
18:                     end for
19:
20:                     // Propagate within each basic block
21:                     BBIn[BB] ← BBIn[BB] ∪ CAPUse[BB] ∪ BBOut[BB]
22:
23:                     // Propagate information in call graph
24:                     if BB contains a call instruction then
25:                         for all functions callee called by BB do
26:                             BBIn[BB] ← BBIn[BB] ∪ funcOut[callee]
27:                             for all basic blocks RB of callee do
28:                                 if RB ends with a return instruction then
29:                                     if BBOut[BB] ⊈ BBOut[RB] then
30:                                         BBOut[RB] ← BBOut[RB] ∪ BBOut[BB]
31:                                         bbChanged ← True
32:                                     end if
33:                                 end if
34:                             end for
35:                         end for
36:                     end if
37:
38:                     if oldIn ≠ BBIn[BB] ∨ oldOut ≠ BBOut[BB] then
39:                         bbChanged ← True
40:                     end if
41:                 end for
42:             until bbChanged = False
43:             changed ← changed ∨ bbChanged
44:         end for
45:     until changed = False
46:     return
47: end function
```

each basic block of a function into the set of capabilities used by the function. It then propagates all of the used capability sets from callees to callers. Since Algorithm 2 iterates until it reaches a fixed point, it can handle recursive functions.

We use the Global Privilege Analysis in the Global Live Privilege Analysis algorithm to reduce the amount of imprecision caused by the lack of context sensitivity in the Global Live Privilege Analysis.

3) Global Live Privilege Analysis: The Global Live Privilege Analysis Algorithm (Algorithm 3) finds the set of live privileges at the entry and exit of each basic block. It is an inter-procedural, context-insensitive, flow-insensitive backwards data-flow analysis. Lines 15 to 18 propagate live privilege information from successor basic blocks to predecessor basic blocks, and lines 20 to 21 propagate live privilege information from the end of a basic block to the beginning of the basic block, adding privileges used within the basic block to the set of privileges live on entry to the basic block.

Lines 23 to 36 propagate live privilege information across functions. Line 26 propagates live privilege sets from callees to callers in the call graph. To reduce the loss of precision caused by context-insensitivity, Algorithm 3 propagates the Global Privilege Analysis results for a function (as opposed to the live privilege sets on entry to the callee function) on line 26. Without the optimization in line 26, if two functions (say F_1 and F_2) called a function F_3, then the live capabilities of F_1 and F_2 would be propagated to each other (via the entry and exit basic blocks of F_3). Lines 27 to 34 propagate live privilege sets from callers to callees. Algorithm 3 does this by propagating privileges that are live at the end of the basic block containing the call instruction to the live privilege sets in all basic blocks of the callee that contain a return instruction.

Algorithm 4 Privilege Removal Transformation Algorithm

```
1:  BBIn: Map from basic blocks to capability set live on entry to block
2:  BBOut: Map from basic blocks to capability set live on exit from block
3:
4:  function PRIVILEGEREMOVAL(BBIn, BBOut)
5:      for all functions f in program do
6:          for all Basic Blocks BB in function f do
7:              if BBOut[BB] ≠ BBIn[BB] then
8:                  Dead ← BBIn[BB] − BBOut[BB]
9:                  Add priv_remove(Dead) to end of BB
10:             end if
11:
12:             for all successors S of BB do
13:                 if BBOut[BB] ≠ BBIn[S] then
14:                     Dead ← BBOut[BB] − BBIn[S]
15:                     Add priv_remove(Dead) to beginning of S
16:                 end if
17:             end for
18:         end for
19:     end for
20:     return
21: end function
```

C. Privilege Removal Transformation

After executing the Global Live Privilege Analysis, AutoPriv inserts calls to `priv_remove` at every program point in which the set of live privileges changes. For each basic block, the live privileges at the start and end of the basic block will either be the same, or the set of live privileges at the end of the basic block will be a proper subset of those live at the beginning of the basic block. The same is true for live privileges at the end of each basic block and the beginning of its successor basic blocks in the control-flow graph.

Our privilege removal transformation, Algorithm 4, looks for all points within a program at which the set of live privileges is reduced between basic block entry and basic block exit and inserts a call to `priv_remove`. Additionally, lines 12 through 17 of Algorithm 4 look for program points in which privileges die between a basic block and its successors and adds a call to `priv_remove` to the beginning of appropriate successor basic blocks.

V. Implementation

a We implemented AutoPriv as a set of new compiler passes for LLVM 3.7.1 [14]. We divided the code into the following smaller LLVM passes:

1) **Split Basic Block Pass:** Transform the program by splitting system calls bracketed by `priv_raise`() and `priv_lower`() calls, as well as calls to internal functions, into separate basic blocks. This simplifies the logic of the local and global privilege analysis passes as well as the privilege removal instrumentation pass.
2) **Local Privilege Analysis Pass:** Compute the local privilege analysis results.
3) **Global Privilege Analysis Pass:** Compute the global privilege analysis results.
4) **Global Live Privilege Analysis Pass:** Compute the global live privilege analysis results.
5) **Privilege Removal Instrumentation:** Add calls to `priv_remove`. Does not instrument signal handlers (which can be called asynchronously) and basic blocks which terminate the program e.g., those ending with an `unreachable` instruction [15].

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

AutoPriv uses a combination of LLVM's built-in call graph analysis pass and the call graph from Data Structure Analysis (DSA) [16] for its inter-procedural analysis passes. The LLVM built-in call graph analysis analyzes direct calls well but provides very pessimistic results for calls using function pointers. AutoPriv uses the LLVM built-in call graph analysis for direct calls. For indirect calls, AutoPriv uses DSA when DSA reports that it has computed a complete set of targets for the call and falls back to the LLVM built-in call graph analysis otherwise.

For our experiments, we refactored the five Linux applications in Table I that run as root to use `priv_raise` and `priv_lower`. Table II shows the number of privilege raising/lowering pairs we added to each program. As Table II shows, even large programs like `sshd` require little refactoring to use privilege bracketing. We also modified `sshd` to use `signal()` instead of `sigaction()` to register signals. This change alleviates the need to use points-to analysis to determine which functions are registered as signal handlers (which AutoPriv does not instrument). Additionally, we modified a function that causes `sshd` to assume that it cannot operate correctly if it does not run as the `root` user.

TABLE I: Programs Used in Experiments

Program	Version	SLOC	Description
passwd	4.1.5.1	51,371	Password change utility
su	4.1.5.1	51,371	Run programs as another user
ping	s20121221	12,001	Send ICMP packets to a remote host
sshd	6.6p1	82,376	Remote login server with encrypted connections
thttpd	2.26	8,367	Web server

TABLE II: Privilege Bracketing Code Changes

Program	Number of Privilege Bracketed System Calls
passwd	29
su	34
ping	6
sshd	59
thttpd	8

VI. Performance Experiments

To evaluate AutoPriv, we studied both its performance when compiling programs and the performance of programs transformed by AutoPriv. Table I shows the programs we chose for our evaluation; we used `sloccount` [17] to measure the number of source lines of code for each program. Because `su`

TABLE III: Compiler Performance

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

and `passwd` belong to the shadow utility suite, we counted the lines of source code in the whole suite. For similar reasons, we counted the lines of source code in the entire iputils suite for ping and the entire OpenSSH suite for sshd. We chose these programs because they run as the `root` user on Unix systems in order to override one or more of the Unix access controls. We created variants of each program that raises and lowers privileges when needed instead of simply switching its effective user ID to/from the `root` user. We used AutoPriv to transform these variants to remove privileges.

We ran our experiments on a machine with an Intel® Core™ i5-6660 processor with 4 cores running at 3.30 GHz and with 16 GB of RAM. The machine ran 64-bit Ubuntu 16.04 Linux.

A. Compiler Performance

To measure analysis time, we used the LLVM `opt` tool to run our global live privilege analysis passes on each program in Table I. We compiled every individual source file into an LLVM bitcode file with -O2 optimization and then linked the separate bitcode files into a single bitcode file with `llvm-link`. We then ran the `opt` tool on the target bitcode file to run the standard -O2 optimization passes for the baseline and the additional AutoPriv passes for AutoPriv; this ensured that AutoPriv analyzed and transformed optimized code.

We used `opt`'s `--time-passes` option to measure the user and system execution time of each pass. We ran this experiment 20 times for each program. Table III shows the average sum of user and system time to run the standard -O2 LLVM optimization passes and the average sum of user and system time to run those same passes and the AutoPriv passes on each program. As Table III shows, AutoPriv induces an average overhead of 19% across our benchmarks on optimization time.

B. Application Performance

To measure the performance overhead that AutoPriv induces on the programs it compiles, we compiled both the original program and the refactored version compiled by AutoPriv at the -O2 optimization level. We compiled both versions using the procedure described in Section VI-A i.e., linking all the bitcode files together and optimizing them with `opt`, to ensure that the only differences were due to privilege manipulation calls. To measure the performance of `passwd`, `su`, and `ping`, we inserted calls to `clock_gettime()` with clock ID set to `CLOCK_MONOTONIC` at the beginning and the end of each program to measure the wall time between the start and end of program execution. To prevent the time for user input from affecting the results, we commented out the call to `getpass()` (which reads in the user's password) in `passwd` and `su` and hard-coded a constant string as the password.

TABLE IV: Runtime of Least Privilege Applications

Program	Version	Average	Std. Dev	Overhead
passwd	Original	36.29 ms	3.75 ms	0.01%
	AutoPriv	36.66 ms	2.94 ms	
su	Original	7.74 ms	0.05 ms	0.01%
	AutoPriv	7.78 ms	0.04 ms	
ping	Original	9,211.49 ms	0.82 ms	0%
	AutoPriv	9,211.42 ms	0.79 ms	

For `passwd`, we ran it with no arguments to change the current user's password. For `su`, we ran it to execute the `echo` command as another user, and we inserted the ending timer before it calls `execve()` to execute `echo`. In our initial experiments, we observed that the execution time of `passwd` and `su` were low due to the small size of the `/etc/passwd` database; we also observed standard deviations above 30%. We therefore ran `useradd` to add 15,000 dummy users and moved the target users' password entries to the end of `/etc/passwd` and `/etc/shadow`. This new password database increased execution time and reduced the standard deviation.

For `ping`, we configured `ping` to ping the localhost machine over the localhost network interface 10 times using the `-c 10` flag for each run of the experiment. Table IV shows the average execution time of repeating `passwd` and `su` 200 times and `ping` 50 times. As Table IV shows, AutoPriv incurs insignificant runtime overheads on these programs; when accounting for standard deviation, AutoPriv incurs no overhead.

To measure the impact of the privilege removal instrumentation on `thttpd` and `sshd`, we measured their average bandwidth on the localhost network interface. We created files containing random contents for the experiments by using Python's `random` library. We used the `random()` function in the library with `time.time()` as the seed for the random number generator. The generated files range from 16 KB to 16 MB. For `sshd`, we transferred one file at a time using the `scp` client. Figures 2 and 3 show the results for `sshd` and `thttpd`, respectively.

For `sshd`, we fetched each file 500 times using `scp`, measured the bandwidth via the `-v` verbose output flag in `scp`, and report the average. For `thttpd`, we used ApacheBench [18] with 10,000 iterations and concurrency level of 32 to measure `thttpd`'s bandwidth. We repeated the experiment 60 times and report the average bandwidth.

As Figure 2 shows, AutoPriv added no overhead to `sshd`; the difference in bandwidth is within a standard deviation. Figure 3 shows the same is true for `thttpd`. The `sshd` server

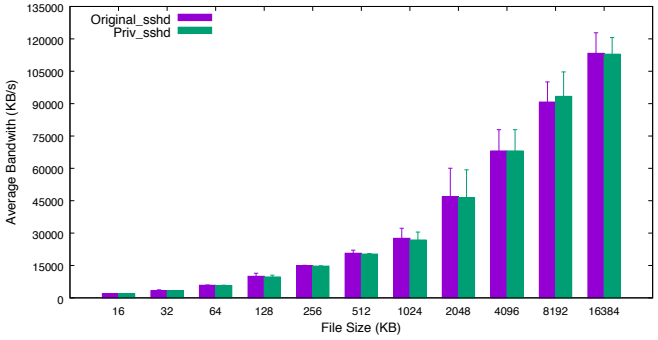

Fig. 2: sshd Performance

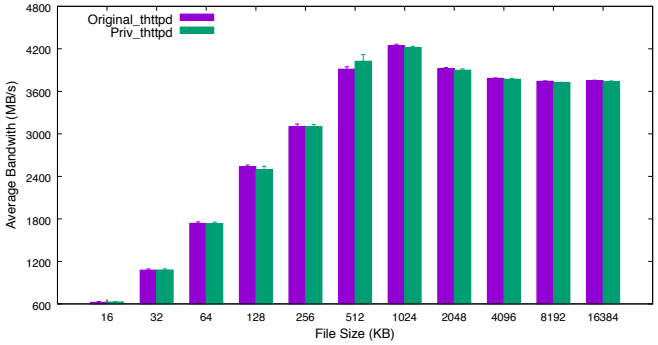

Fig. 3: thttpd Performance

uses most of its privileges when logging a user in to start a new SSH session; `thttpd` uses all of its privileges before accepting network connections. Most of the execution time is spent transferring data; by that point in time, these servers have finished manipulating privileges and therefore incur no measurable overhead.

VII. Related Work

Several approaches allow programmers to manually modify their programs to prevent privilege escalation attacks. The Compartmented Mode Workstation (CMW) [8] introduced fine-grained privileges and mechanisms for raising and lowering privileges; Linux capabilities borrow ideas from the CMW. While the original CMW requires programmers to modify their programs manually, AutoPriv automatically modifies programs to permanently remove privileges.

Privilege separation [19] partitions an application into multiple processes such that only one process runs with root privilege; this privileged process provides a restricted interface to the other application processes for performing operations as the root user. Privtrans [20] is a compiler that simplifies the creation of privilege-separated programs. However, the programmer must design the interface to the privileged process in a way that reduces the attack surface and annotate the program to communicate this information to Privtrans; the compiler only automates the mechanical process of splitting the application into multiple processes. We believe that we could combine AutoPriv with privilege separation to provide security that neither approach can achieve individually.

Capsicum [21] adds new kernel features and library support to Unix to compartmentalize, or privilege separate, applications. Capsicum provides a capability mode that processes can enable for themselves. Once enabled, capability mode prevents a process from accessing objects in global OS namespaces e.g., files, and extends file descriptors with capabilities that constrain the operations that a process can perform on those file descriptors. Since access to global namespaces is restricted once a process enters capability mode, the programmer must modify an application so that either a non-capability mode process passes new file descriptors to sandboxed processes via Unix domain sockets or all file descriptors to be used by a sandboxed process must be obtained before the sandboxed process enters capability mode. While Capsicum lacks a compiler analysis to help remove capabilities when no longer needed, we believe AutoPriv could be expanded to perform this task.

Capweave [22], given a security policy that describes when and which capabilities a program should hold during its execution, instruments the program to use the Capsicum primitives in order to enforce the policy. Although Capweave's compartmentalization is automatic, it still requires a programmer or user to manually specify the security policy. In contrast, AutoPriv enforces a simple policy (that of removing dead privileges) that is hard-coded into the AutoPriv compiler.

Systrace [23] restricts the values that applications can use as system call arguments. It creates its policies either automatically via dynamic tracing of program execution or interactively with the user's input [23] while AutoPriv only requires programmers to privilege bracket system calls. Systrace also allows users to specify which system calls need to run with root privilege, allowing the rest of the application to execute as a regular user [23]. Like Linux capabilities, Systrace can limit the damage done by an exploited application. However, Systrace is susceptible to time-to-check-to-time-of-use attacks [24]; Linux capabilities (and AutoPriv) are not.

M. Rajagopalan et al. [25] use compiler techniques and kernel runtime checks to enforce system call policies. Their system uses a trusted program installer that automatically finds system calls in the application binary and augments them with arguments that specify the policy that the system call should satisfy as well as a Message Authentication Code that the kernel uses at runtime to verify the integrity of the policy and the system call arguments. The policies that M. Rajagopalan et al. [25] enforce restrict the values that system calls arguments can take. While this approach verifies system call arguments, an attacker can still execute privileged system calls repeatedly since their system does not remove unnecessary privileges.

Protego [26] modifies the AppArmor Linux Security Module to enforce the security policies enforced by setuid root binaries and enforces these policies with fewer lines of code. However, Protego isn't designed to protect network servers that run as root [26] while AutoPriv can reduce the privilege use of both setuid root programs and privileged server programs.

Koved et al. [27] present an algorithm that automatically determines the access rights needed by a Java program. Both their algorithm and ours propagate information on required access rights i.e., privileges, upward through a static representation of the program's control flow. In addition to analyzing the use of operating system privileges, the key difference between their work and ours is that our algorithm searches for program points at which the required set of privileges changes; these points are where a program can permanently and safely remove a privilege (or access right) that it no longer needs.

VIII. Conclusions and Future Work

This paper describes the AutoPriv compiler which analyzes privilege use in applications and transforms them to permanently remove privileges when no longer needed. AutoPriv incurs, on average, 19% overhead during optimization and induces practically no overhead in the programs that it compiles.

Several interesting directions exist for future work. First, we can port AutoPriv to different operating systems to determine if our approach generalizes to other privilege models. For example, we could enhance AutoPriv to analyze Capsicum programs [21] to find program points at which to remove capabilities from file descriptors.

Second, we will investigate whether improvements to our compiler analysis will yield better results. AutoPriv currently assumes that all local control flow paths are feasible and uses a conservative call graph. Using path-sensitive analysis may allow privileges to be removed earlier.

Third, we can combine AutoPriv with automated privilege separation [20]. Since AutoPriv calculates the code regions in which privileges are live, it can calculate the different combination of privileges that the program can use and which system calls need them. This information could be used to guide privilege separation so that different processes have different privileges, ensuring that no one process has enough privileges to be equivalent to a process running as `root`.

Acknowledgements

We thank the anonymous reviewers for their feedback. This work was supported by NSF Award 1463870.

Open Source

We open-sourced our AutoPriv compiler. The code is available at https://github.com/jtcriswell/AutoPriv/tree/AutoPriv.

References

[1] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[2] D. P. Bovet and M. Cesati, *Understanding the LINUX Kernel*, 2nd ed. Sebastopol, CA: O'Reilly, 2003.

[3] M. E. Russinovich and D. A. Solomon, *Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server(TM) 2003, Windows XP, and Windows 2000 (Pro-Developer)*. Redmond, WA, USA: Microsoft Press, 2004.

[4] J. Mauro and R. McDougall, *Solaris Internals: Core Kernel Architecture*. Prentice Hall PTR, 2000.

[5] S. E. Hallyn and A. G. Morgan, "Linux capabilities: Making them work," in *Proceedings of The Linux Symposium*, Ottawa, Canada, July 2008.

[6] A. One, "Smashing the stack for fun and profit," *Phrack*, vol. 7, November 1996. [Online]. Available: http://www.phrack.org/issues/49/14.html

[7] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information Systems Security*, vol. 15, no. 1, pp. 2:1–2:34, Mar. 2012.

[8] J. L. Berger, J. Picciotto, J. P. L. Woodward, and P. T. Cummings, "Compartmented mode workstation: Prototype highlights," *IEEE Trans. Softw. Eng.*, vol. 16, no. 6, pp. 608–618, Jun. 1990.

[9] M. K. McKusick and G. V. Neville-Neil, *The Design and Implementation of the FreeBSD Operating System*. Pearson Education, 2004.

[10] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *14th USENIX Security Symposium*, August 2004, pp. 177–192.

[11] Solar Designer, "return-to-libc attack," August 1997, http://www.securityfocus.com/archive/1/7480.

[12] Argus Systems Group, Inc., "Security features programmer's guide," Savoy, IL, September 2001.

[13] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1999.

[14] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *Proceedings of the Conference on Code Generation and Optimization*, San Jose, CA, USA, Mar 2004, pp. 75–88.

[15] C. Lattner *et al.*, "LLVM Language Reference Manual." [Online]. Available: http://releases.llvm.org/3.7.1/docs/LangRef.html

[16] C. Lattner, A. D. Lenharth, and V. S. Adve, "Making context-sensitive points-to analysis with heap cloning practical for the real world," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, USA, June 2007, pp. 278–289.

[17] D. A. Wheeler, "SLOCCount," 2014. [Online]. Available: http://www.dwheeler.com/sloccount/

[18] "Apachebench: A complete benchmarking and regression testing suite. http://freshmeat.net/projects/ apachebench/," July 2003.

[19] N. Provos, M. Friedl, and P. Honeyman, "Preventing privilege escalation," in *Proceedings of the 12th USENIX Security Symposium*, 2003.

[20] D. Brumley and D. Song, "Privtrans: Automatically partitioning programs for privilege separation," in *13th USENIX Security Symposium*, 2004, pp. 57–72.

[21] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway, "Capsicum: Practical capabilities for UNIX," in *Proceedings of the 19th USENIX Conference on Security*, ser. USENIX Security'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 3–3.

[22] W. R. Harris, S. Jha, T. Reps, J. Anderson, and R. N. M. Watson, "Declarative, temporal, and practical programming with capabilities," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, ser. SP '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 18–32.

[23] N. Provos, "Improving host security with system call policies," in *12th USENIX Security Symposium*, August 2003.

[24] R. N. M. Watson, "Exploiting concurrency vulnerabilities in system call wrappers," in *Proceedings of the First USENIX Workshop on Offensive Technologies*, ser. WOOT '07. Berkeley, CA, USA: USENIX Association, 2007, pp. 2:1–2:8.

[25] M. Rajagopalan, M. Hiltunen, T. Jim, and R. Schlichting, "System call monitoring using authenticated system calls," *Dependable and Secure Computing, IEEE Transactions on*, vol. 3, no. 3, pp. 216–229, July 2006.

[26] B. Jain, C.-C. Tsai, J. John, and D. E. Porter, "Practical techniques to obviate setuid-to-root binaries," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys '14. New York, NY, USA: ACM, 2014, pp. 8:1–8:14.

[27] L. Koved, M. Pistoia, and A. Kershenbaum, "Access rights analysis for Java," in *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '02. New York, NY, USA: ACM, 2002, pp. 359–372.