Future Generation Computer Systems 135 (2022) 105–113

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

The devil is in the detail: Generating system call whitelist for Linux seccomp

Yunlong Xing [a,c], Jiahao Cao [b,c], Kun Sun [c], Fei Yan [a,*], Shengye Wan [d]

[a] School of Cyber Science and Engineering, Wuhan University, Wuhan, China
[b] Tsinghua University, Beijing, China
[c] George Mason University, Fairfax, VA, USA
[d] College of William & Mary, Williamsburg, VA, USA

ARTICLE INFO

Article history:
Received 2 October 2020
Received in revised form 11 April 2022
Accepted 16 April 2022
Available online 12 May 2022

Keywords:
System call restriction
Attack surface reduction
Whitelist generation
Software security
System enhancement
Static analysis

ABSTRACT

The system calls provide the main interface for user processes to request the kernel services, however, for any specific process, most of them will not be needed. If a user process is compromised, those unnecessary system calls can be abused to attack the kernel and the other processes. To migrate this problem, the seccomp mechanism has been merged into the Linux kernel to limit the available system calls according to a system call whitelist. However, it is still a challenge to automatically and effectively generate a minimal but complete system call whitelist for a specific user process. In this paper, we develop a toolkit named TAILOR that mainly relies on the static analysis to generate a mapping table for the standard library from the library functions to their corresponding system calls based on the source code analysis. Then for any application that invokes system calls via the standard library, we can just compare the called library functions in the application with the mapping table to obtain required system calls. TAILOR solves the problems during source-level standard library analysis, which consist of the difficulty in macro function identification, unchained calling tree caused by macro aliases, and the difficulty in identifying the function scope via static analysis. Our experiments on 50 popular general terminal commands show that our tool can reduce 88% system calls for them and block about 74% potential vulnerabilities from malicious system calls.

1. Introduction

Linux introduces strong isolation boundaries and segregates the virtual memory into user space and kernel space to provide memory protection and hardware protection from malicious or errant software behavior [1]. All privileged functions, such as process scheduling, IPC (Inter-process communication), memory management, and network management et al., can only be executed in the kernel space. The only interface for switching from user space to kernel space is via the system call mechanism.

The latest stable Linux kernel (v5.18) provides about 350 system calls. However, for any specific user process, even if it only requires a small subset of the system calls for its normal running, it has the potential to invoke all system calls. When a user process is compromised and tricked into making vulnerable system calls, it may compromise the entire system by breaking into the kernel space [2–4]. To enhance system security, the seccomp [5] mechanism has been merged into the Linux kernel to restrict what system calls can be made by a user process. The mechanism has been widely used in academic research and industrial products [6,7]. However, it is still a challenging problem on how to automatically decide the system call whitelist for seccomp to configure the filter rules for a given process.

To obtain the required system calls for a specific user process, the state-of-the-art solutions rely on dynamic analysis to capture system calls when the target process is being dynamically traced [8–10]. Although they can easily find most system calls, they have several limitations. First, they cannot guarantee to identify all required system calls due to the incomplete coverage of dynamic process execution. Later, a missing system call may hang the process being protected by seccomp. Second, it is time-consuming to dynamically trace different paths of a process. Particularly, the user process usually invokes system calls by calling various wrapper functions in different libraries, which significantly increases the tracing complexity.

In this paper, we develop a toolkit called TAILOR (sysTem cAll whIteList generatOR) that mainly relies on the static analysis to automatically and efficiently generate a complete system call whitelist for seccomp, which can constrain the available system calls for a any given process running on Linux machine.

* Corresponding author.
E-mail address: yanfei@whu.edu.cn (F. Yan).

https://doi.org/10.1016/j.future.2022.04.016
0167-739X/

Since most user processes invoke system calls via the standard library, we develop a *function-related system call profiler* and an *environment-related system call profiler* that work together to build a mapping table from library functions to the corresponding system calls. The function-related system call profiler is responsible for collecting the system calls invoked by each library function via statically analyzing the source code of library. The environment-related system call profiler focuses on profiling the system calls required to set the running environment (e.g., memory loading) for each library function. Moreover, it includes a *library function profiler* to collect the library functions that are called by a target program. By matching the called library functions with the mapping table, we can obtain the system call whitelist required by the target program via the libraries. In addition, we can obtain the whitelist of system calls invoked by directly calling assemble instructions and system call functions via static analysis. After merging these two whitelists, we can generate the final system call whitelist.

The function-related system call profiler performs a static analysis over the library source code to output the system calls invoked by each library function. The whole process is divided into three steps: *building a function call tree (FCT) for each file*, *connecting all FCTs into a function call graph (FCG)*, and *obtaining the mapping between library functions and system calls*. In the first step, we conduct a lexical analysis and create a deterministic finite automaton to help identify all function names. After that, we distinguish all caller functions and callee functions by recognizing function call relationships. Then we chain function call relationships to build a FCT for each file. In the second step, we merge the same nodes on different FCTs to build a FCG. Finally, we traverse the FCG to obtain the corresponding system calls for each library function, i.e., function-related system calls.

Besides function-related system calls obtained by analyzing source codes, there are some other system calls required for setting the running environment, such as memory loading, file accessing, and status reading. As these system calls are invoked at run time, they cannot be obtained by analyzing source codes. However, when any of these system calls is blocked, we cannot run programs properly. Thus, we build a running environment for each library function and track them to obtain the environment-related system calls. Particularly, our study demonstrates that the number and type of environment-related system calls remain stable. Therefore, we can obtain the complete environment-related system calls for each library function. By obtaining function-related system calls and environment-related system calls, TAILOR builds a mapping table from library functions to their corresponding system calls. Note that the mapping table only needs to be constructed once for a specific-version library.

After building the mapping table, TAILOR leverages a library function profiler to obtain the library function list called by the target program. It takes source codes or ELF files as inputs to obtain the called function list. Afterwards, the profiler identifies the called library function list according to the library functions in the previously generated mapping table. Finally, it combines the library function list called by the target program with the mapping table to get the library-related system call whitelist. Moreover, as there are a few cases where applications may directly invoke system calls, we identify these system calls with specific identifiers. They will be merged into the final system call whitelist.

We conduct extensive experiments to evaluate the performance of our tool. We first measure the number of system calls required for a library function. The experimental results show that most of the library functions invoke less than 20 system calls. Moreover, we choose 50 general commands and calculate the called library functions and the required system calls. Our experiments show that over 80% of testing commands only require less than 100 library functions and all these testing commands call 76 library functions on average. Also, the number of invoked system calls for these commands is between 35 and 75 and over 90% commands invoke less than 50 system calls. Thus, the system call reduction rate can reach 88% on average. The results demonstrate that library functions only need a small number of system calls. To further verify that TAILOR can reduce the attack surface with the system call whitelist, we collect all Common Vulnerabilities and Exposures (CVE) related to system calls in recent five years. The results show that 74% of vulnerabilities can be blocked for these commands by setting the system call whitelist.

In summary, we make the following contributions:

- We design a toolkit called TAILOR to generate the system call whitelist for a specific user process. It can effectively reduce the attack surface by removing unnecessary system calls that may be misused by malicious attackers.
- We present the system call profiling mechanism that can build a mapping table between all library functions and system calls.
- We evaluate the performance of TAILOR on 50 general terminal commands. The experiments show that our toolkit can effectively reduce the number of unnecessary system calls and prevent existing vulnerabilities reported in CVE.

2. Background

In this section, we briefly introduce the necessary background on system calls, library functions, and secure computing mode.

2.1. System calls and library functions

Linux provides three methods for user programs to invoke system calls, i.e., calling system call functions, calling assemble instructions, and calling library functions [11]. First, it can directly call a system call function, such as the function `syscall` with a parameter denoted the name of an expected system call. Second, the program can embed assemble instructions to invoke a system call through passing a system call number to the `EAX` or `RAX` register and triggering a software interrupt to switch into kernel space. However, there are two main disadvantages for the above two methods. First, they may introduce frequent context switching between user space and kernel space when directly invoking system calls. For example, when using the `write` system call to output a string to a file, the program must switch to kernel space and back to user space for each character. This switching can incur huge overhead for programs. Second, as system calls vary in different operating systems, directly invoking system calls cannot be easily ported to different operating systems.

Instead, most applications invoke system calls by calling the library functions provided by standard library such as GNU C Library (glibc), which provides a universal, efficient, and secure way to invoke system calls on different operating systems. For example, when using the `printf` library function to output a string, all the characters are first buffered and then are flushed to a file using the `write` system call. In such a case, there is only one switch from user space to kernel space, which significantly reduces the switching overhead [12]. Moreover, library functions are system-independent and make low-level services transparent to developers. Due to the efficiency and convenience of library functions, almost all programs use library functions to invoke system calls [13,14].

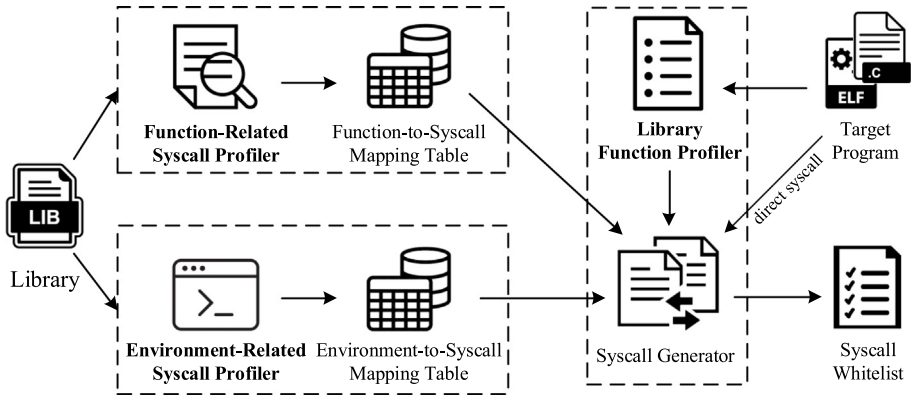


Fig. 1. TAILOR architecture.

2.2. Secure computing mode

Secure computing mode (`seccomp`) is a security mechanism that can restrict the system calls available for a process. Seccomp works in either strict mode or filter mode [15]. In the strict mode, a user process can only invoke 4 system calls, i.e., `read`, `write`, `exit`, and `sigreturn`. Any other system calls tried by this process will be blocked. In the filter mode, a configurable policy containing available system calls can be specified for a process. To use this mode, three parameters should be provided to set the filter. The first parameter `SECCOMP_SET_MODE_FILTER` sets the filter mode, the second parameter sets `flag` information, such as process synchronization and logging, and the third parameter `args` points to available system calls defined by a Berkeley Packet Filter (BPF) [5]. Although the first version of seccomp had been merged into the Linux kernel in 2005, there still lacks a reliable toolkit to automatically generate available system call whitelist for given programs.

3. System overview

Currently, widely exposed system interfaces have become a critical security challenge [16–21] to operating systems. For example, in the latest Linux kernel (v5.18), about 350 system calls are provided by default for the user-privileged untrusted process requiring kernel-privileged services. Nevertheless, most user processes only need a small portion of the whole system calls to accomplish their tasks. Consequently, if any user process is compromised to trigger vulnerable system calls, related kernel services may get attacked. To solve this problem, we develop a system toolkit called TAILOR, which automatically customizes the system call whitelist for untrusted user-privileged programs. With the assistance of the whitelist, user programs can be prevented from calling unnecessary system calls, and hence the attacking surface of the system kernel can be effectively reduced.

The architecture of TAILOR is shown in Fig. 1. To profile all system calls required by the target program, we provide two profilers to analyze the relationship between library functions and system calls: (i) a function-related system call profiler to output the system calls invoked by each library function, and (ii) an environment-related system call profiler to generate the system calls invoked when the library functions are being loaded and executed. By merging the function-related system calls and the environment-related system calls, we can get a mapping table from the library function to corresponding system calls. As almost all programs invoke system calls via library functions, we design a library function profiler to collect all library functions of the target program. By comparing these library functions with the mapping table, library-related system calls of the target program can be generated. Moreover, as there are a few cases where programs directly invoke system calls through inline assembly instructions or system call functions, TAILOR identifies these system calls with specific identifiers. Finally, TAILOR obtains the complete system call whitelist by merging the library-related system calls with the direct system calls.

In the following sections, we detail the design of three main components of TAILOR, namely, the function-related system call profiler, the environment-related system call profiler, and the library function profiler.

4. Function-related system call profiler

The function-related system call profiler is responsible for building the mapping table from library functions to function-related system calls. Fig. 2 shows the process of obtaining system calls. For each file in a library, we build a Function Call Tree (FCT) to denote function call relationships in a file. Afterwards, we combine FCTs in all files to build a directed acyclic Function Call Graph (FCG), which denotes all function call relationships between library functions. Moreover, as the system calls will not call back other library functions, the system calls always appear at the ending nodes in an FCG. Therefore, a mapping table from library functions to system calls can be built by traversing all paths in an FCG.

4.1. Building an FCT for each file

An FCT is a tree structure where each node is a library function that has its callee functions as child nodes. It can clearly represent the caller/callee relationships between functions, ignoring some unnecessary details like variable declarations and operations.

Identifying Function Names. In programming languages, the definition, declaration, and calling of functions have a fixed format. For example, in C language, the definition of functions is: *dataType functionName(parameter list){/*body*/}*. In this expression, the `dataType` will be chosen from the keyword list, and the `parameter list` can be set void or a list of identifiers with keywords ahead. Similarly, when calling a function, the format of the callee functions will be like *functionName(parameter list)*. The `functionName` above is an identifier. Thus, it will obey the naming rule for the identifier.

To identify the functions in each file, we design a lexical analyzer based on `clang` [22], a popular LLVM compiler front-end, to scan all character sequences and transform them into tokens. Here, scanning character sequences is based on finite-state automation. In many cases, we can infer the type of tokens based on the first non-blank character. The subsequent characters can be processed one by one until a character that does not belong to the

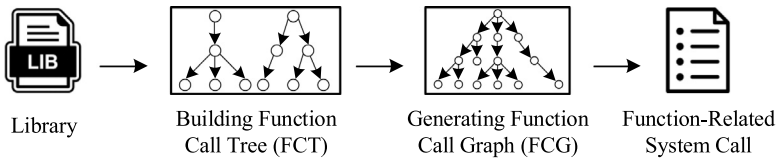


Fig. 2. Obtaining function-related system calls.

type of the tokens appears. Then tokenization splits the tokens and classifies them with corresponding classification marks, such as `identifier` and `comma`. Finally, we match the identifiers with function definitions and calling format. The classification marks of function identifiers are changed to `function`.

Recognizing and Chaining Function Call Relationships. After identifying function names, we can recognize the function call relationships and chain them together to form FCTs. To identify the function call relationship, we first distinguish the caller functions from the callee functions. In C language, the function body cannot contain the definition of another function. In other words, the definition of functions cannot be nested. Thus, if we cannot find the keywords closest ahead of a `function` mark, this function should be considered as a callee function. As for a caller function, it should have a keyword ahead of the `function` mark and a left-parentheses mark followed to show the beginning of the function body. We consider a caller function as a parent node and consider all functions inside the caller function body as the callee functions that are denoted as child nodes.

4.2. Generating an FCG via combining all FCTs

After obtaining FCTs in library files, we connect them into a directed acyclic FCG. The key technique to connect all FCTs is to merge the same nodes. For example, the function `printf` implements by calling the function `vfprintf`. Meanwhile, the function `sprintf` also calls `vfprintf`. If we merge these two FCTs by the shared node `vfprintf`, we can build the connection between different functions while reducing repeating analysis of the same node. When we merge FCTs, there are two special cases that should be handled carefully, i.e., function alias and function call scope.

Processing Function Alias. One common situation of libraries' implementations is that multiple functions may share the same functionality with different function and parameter names. Such function alias can cause an FCG unconnected. For example, when analyzing `printf`, there is a statement `#define PUT(f, s, n) _IO_sputn(f, s, n)`, which replaces `_IO_sputn` with `PUT`. When merging FCTs for `printf`, it may contain two separate call trees. To obtain a complete call graph, we merge all function aliases with the same functionality into the same tree. We also observe that when there is a function alias, there are usually multiple aliases followed. Therefore, when connecting FCTs, we need to link several function alias in a chain. To eliminate the impacts of function alias when building an FCG, we build a set of function alias for each chain of function alias and choose one function name in the set to replace all aliased function names in the same set.

Handling Function Call Scope. When there are multiple implementations of the same callee function, we should choose the appropriate one to build the FCG. For example, if function `A` calls `B`, `B` may have different implementations in three different places, namely, in the same file, in the same directory, and out of the directory. Similar to how a compiler deals with this situation, we first find the implementation of the callee function `B` in the same file. After that, we search the callee function in the header files of the same directory. Finally, we search its implementation out of the directory. Once we find the implementation in a place, we will choose it and stop the search.

Algorithm 1 Merging Two Function Call Trees

Input:
$G(V, E)$, $G'(V', E')$
$v'_i, v'_j \in V'$, *two node of* E'_i
Output:
$G(V, E)$ *after merging with* $G'(V', E')$

```
1:  i ← 0
2:  while E'_i ≠ null do
3:    if E'_i ∉ E then
4:      if only one tree rooted by v'_i then
5:        E ← E + E'_i
6:        V ← V + v'_i + v'_j
7:      else
8:        add the highest priority tree
9:      end if
10:   else
11:     merge trees rooted by v'_i
12:   end if
13:   i ← i + 1
14: end while
15: if function alias exists then
16:   put all aliased functions in a set
17:   use starting node to express the chain
18:   connect function node with alias
19: end if
```

Merging FCTs into one FCG. The merging algorithm is shown in Algorithm 1. The input of this algorithm includes two trees, G and G' with node sets, V and V' and edge sets, E and E', as well as the ith edge, E'_i, in G' and its two nodes, v'_i and v'_j. Line 1 initializes auxiliary variable i to traverse all edges in E'. The main loop is from Line 2 to Line 14. Line 3 determines whether the ith edge, E'_i, is in the set E. If not and only one tree is rooted by v'_i, we add the edge E'_i to E and add the nodes v'_i and v'_j to V, as shown in Line 3 to Line 6. If there are several trees rooted by v'_i, we choose the highest priority tree according to the function call scope in Line 7. If the ith edge E'_i is in the set E, we should merge trees rooted by v'_i with G, as shown in Line 11. In Line 13, it increases variable i to continue the next loop. Between Line 15 to Line 19, we check if there are function alias, if so, we connect those function alias in one set and choose the first aliased function to replace all other function alias. After doing that, we can obtain a new FCG $G(V, E)$ after merging with $G'(V', E')$.

4.3. Obtaining function-related system call mapping table

As system calls do not call any other library functions, system calls are at the leaf nodes in an FCG that we build. To obtain the mapping relation between library functions and their corresponding system calls, we take the library function name as the starting node and use the depth-first algorithm to traverse the graph. Moreover, to guarantee each node will be searched only one time, we set a flag for all nodes to record the access information. If a

Algorithm 2 Retrieving System Calls from Call Graph

Input:
$G(V, E)$; *start*, *node to be traversed*
Output:
system calls, *corresponding to start node*

```
1: nextnode ← null
2: path −> next ← start
3: visited −> next ← start
4: stack −> next ← start
5: while stack ≠ null do
6:   current ← get stack top
7:   adj ← get adjacency list of current
8:   if adj ≠ null then
9:     for reading a node in adj do
10:      if the node has not accessed then
11:        nextnode ← this node
12:        break
13:      end if
14:    end for
15:  end if
16:  if nextnode ≠ null then
17:    path −> next ← nextnode
18:    visited −> next ← nextnode
19:    stack −> next ← nextnode
20:  else
21:    if system call node then
22:      record the syscall name
23:    end if
24:    stack −> next ← null
25:  end if
26: end while
```

node has been accessed, we can reuse the result corresponding to this node. After traversing all library functions, we can obtain a mapping table with two entries, namely, library functions and its corresponding function-related system calls. Due to space limitations, we present the detailed traversing algorithm as Algorithm 2.

The input of this algorithm is the FCG G, with node set V and edge set E, and the node to be traversed `start`. Line 1 to Line 4 initialize all auxiliary variables. Among them, the `nextnode` stands for the unvisited adjacent node for the current visited node, initialized to `null`, the `path` represents the paths of the adjacent nodes, the `visited` denotes visited nodes, and the `stack` means an first-in-last-out list. All the last three variables are initialized to the `start` node. The main loop is from Line 5 to Line 26. Line 6 and Line 7 initialize the variables `current` and `adj` to get the value of the stack top and the adjacency list of `current`. If the adjacency list is not null, we read a node in `adj`. If the node has not been accessed, we assign this node the variable `nextnode` and jump out this loop, as shown in Line 8 to Line 15. Then we put the `nextnode` to the stack top and continue analyzing its adjacency list, as shown in Line 16 to Line 19. If the `nextnode` is equal to null, meaning that this node is a leaf node, then we determine whether the current node is a system call. The main method is based on the observation that the library function will invoke system calls by several specific macros, such as `INLINE_SYSCALL`, `SYSCALL_NOCANCEL`, in which the first parameter is the system call name. After identifying these macros in the lead node, we record the system call in the first parameter, as shown in Line 21 to Line 23. After that, we have reached the end of this branch, so we will pop the top element of the stack and continue searching for the next adjacent node, as shown in Line 24. By finally emptying the stack, we can obtain the system calls corresponding the `start` node.

5. Environment-related system call profiler

When running a program, there will be specific system calls needed to set the running environment, such as, reading file status, initializing memory, and setting stacks. These environment-related system calls cannot be obtained by analyzing the code of library functions. Thus, we track the program executing to obtain the environment-related system calls for the library functions. Particularly, our experiments show that the number and types of system call setting the running environment for a library function are relatively fixed. Thus, we can program and compile each library function into an executable file and then track the environment-related system calls during the program execution. Fig. 3 shows the two main steps to obtain environment-related system calls, i.e., building the running environment and tracking system calls.

Building Running Environment. To track the environment-related system calls of every library function, we need to prepare the running environment for them correspondingly. We first insert strings printing with unique characteristics both at the beginning and the ending of a program and all traced system calls ahead of the beginning point and after the ending point are considered as the environment-related system calls. In some cases, the built-up running environment should take the context dependency into consideration. For example, when closing a file using `close`, it is required that a corresponding function `open` has been invoked for that file. We also observe that these dependent functions are clearly defined in the Linux manual pages [23] and can be used cooperatively. Therefore, we build a table to collect all binding function pairs, whenever there is one function in the pair called, we will insert all its dependent functions into the program.

Tracking System Calls. After building the running environment, we can track them to get the system calls required at run time. The main tool we use for tracking is `strace` [24], which is an open-source tracking tool for Linux processes. It can be used to obtain system calls required for a process and count the number of system calls with option `-c`. By tracking and obtaining library functions' environment-related system calls, we can build a mapping table between library functions and environment-related system calls. The mapping table can help profile system calls for different target programs.

6. System call whitelist generation

To generate the system call whitelist for a given program, we first develop a library function profiler to obtain the library functions called in each program. Next, we develop a system call generator to obtain the list of system calls by mapping the library functions into the library-to-syscall mapping table. We also develop a direct syscall profiler to collect the set of system calls that are directly invoked by the program. Finally, we obtain the complete system call whitelist by merging the lists of library-related system calls and direct system calls.

Library Function Profiler. It is responsible for extracting all called library functions in each program. The profiler takes source codes or ELF files as the inputs. When the source code of the program is available, the profiler utilizes an existing tool `cflow` [25] to obtain all functions presented in the code. By checking if a function is in the lib-to-syscall mapping table, we can decide if it is a library function. Given an ELF file, the profiler exploits the ELF symbol table, which contains all global variables and function information, to derive the called functions. Specifically, the profiler first uses the tool `readelf` [26] to acquire a readable ELF symbol table for the ELF file. After that, the profiler extracts the library functions according to the lib-to-syscall mapping table. Fig. 4 shows an example of the ELF symbol table. We can see that

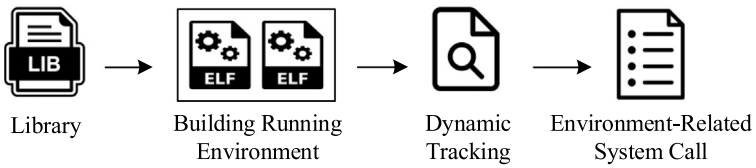


Fig. 3. Obtaining environment-related system calls.

```
Symbol table '.symtab' contains 63 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000238     0 SECTION LOCAL  DEFAULT    1
     2: 0000000000000254     0 SECTION LOCAL  DEFAULT    2
                                 ...
    27: 0000000000000570     0 FUNC    LOCAL  DEFAULT   14 deregister_tm_clones
    28: 00000000000005b0     0 FUNC    LOCAL  DEFAULT   14 register_tm_clones
    29: 0000000000000600     0 FUNC    LOCAL  DEFAULT   14 __do_global_dtors_aux
                                 ...
    47: 00000000000006e4     0 FUNC    GLOBAL DEFAULT   15 _fini
    48: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND printf@@GLIBC_2.2.5
    49: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@@GLIBC_
```

Fig. 4. An example of ELF symbol table.

for any derived ELF symbol table, the profiler can directly identify all called functions with the `Type` as `FUNC` and further get the functions' names as shown in the column of `Name`.

System Call Generator. After obtaining all the called library functions of a program, we cross-check the identified function list with the library-to-syscall mapping table. Therefore, we can identify all library-related system calls for the target program. Note that even if the program does not invoke any library-related functions, several fundamental system calls are still needed as the baseline support for executing any program. In this case, the generator considers these system calls as a subset of environment-related system calls and includes them in the final system call whitelist. Considering a few programs invoke system calls directly through inline assembly instructions or system call functions, we use keywords to retrieve this kind of invoking, such as `SYSENTER` and `int 0x80`. As the EAX or RAX register stores the corresponding system call number, we track back the value of EAX or RAX register to obtain the name of the invoked system call. The final syscall whitelist consists of both library-related system calls and directly invoked system calls. By setting the system call whitelist to the filtering rule of `seccomp`, we can restrict the available system calls for a target program and thus reduce the attack surface of misusing system calls.

7. Evaluation

We conduct extensive experiments to evaluate the performance of our toolkit TAILOR. All experiments are conducted on a computer with an Intel Xeon E5-2620 2.4 GHz 12-core processor and 16 GB of RAM. It runs Ubuntu 18.04 with the kernel version 4.15.0. The GNU C Library (glibc) version is 2.29. We first measure the number of system calls for different library functions in glibc. And we select 50 popular terminal commands and calculate the called library functions and the invoked system calls for these commands. To measure the correctness and the effectiveness of this tool, we compute the vulnerability defense rate for these testing commands after setting the system call whitelist.

7.1. Number of system calls invoked by one library function

We first measure the number of system calls that may be invoked by a library function in glibc. The total number of library functions is 1232. Fig. 5(a) shows the cumulative distribution function (CDF) of the number of system calls for each library function. We can see that in the testbed, one glibc library function invokes at least 11 system calls, i.e., `execve`, `fstat`, `access`, `close`, `read`, `openat`, `arch_prctl`, `mmap`, `mprotect`, `munmap`, and `brk`. Among them, `execve` executes the file; `access` and `fstat` access and get the status of the library file; `arch_prctl` sets the thread status; `read`, `openat` and `close` open and read the library file; `brk`, `munmap`, `mmap`, and `mprotect` initialize and allocate the memory. All of them are related to setting a running environment even if the function does not invoke any function-related system calls. Moreover, we see that the functions in glibc can invoke 39 system calls at most. For about 90% of library functions, the number of invoked system calls is less than 20.

7.2. System calls required for general commands

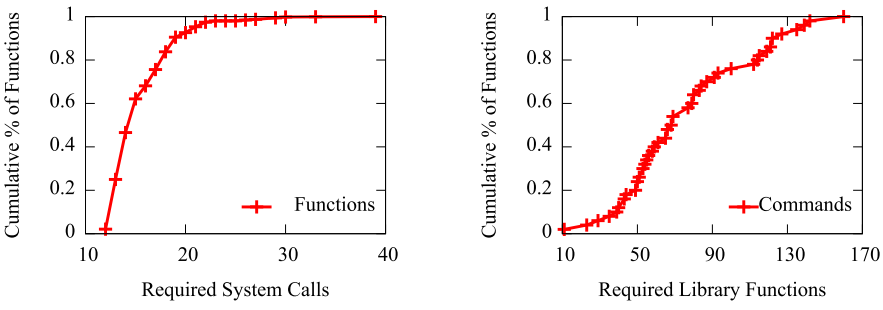
Then we select 50 popular general commands, such as `cat`, `dir`, `chmod`, and the detailed information about the testing set is shown in Appendix. After comparing the called functions with the mapping table, we can obtain the called library functions for each command. The statistical data is shown in Fig. 5(b), we can see that one command calls 11 library functions at least and 160 library functions at most. Over 80% of testing commands only require less than 100 library functions and all these testing commands call 76 library functions on average.

7.3. System call reduction

For all these commands, they can possibly invoke all the system calls by default, and the total number of system calls is 333 in 64-bit compilation mode. Fig. 6(a) shows the cumulative distribution function of required system calls for these testing commands, and we can see that our tool can significantly reduce the system call whitelist. Particularly, the number of invoked system calls is between 35 and 75 and over 90% commands invoke less than 50 system calls. Thus, the system call reduction rate can reach 88% on average.

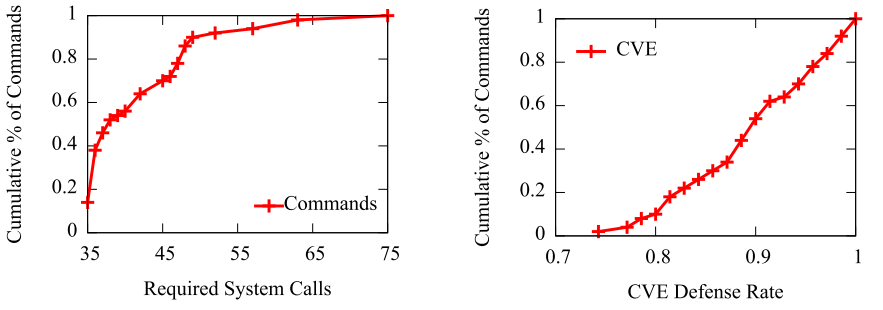
To verify that commands can properly work with the reduced system calls, we use `seccomp` to enforce rules for each command so that it can only use the system calls in the whitelist. We execute each command and manually trigger its various functions to traverse all branch conditions as much as possible. We find that all the commands can properly work after enforcing the whitelist. It means TAILOR does not miss the necessary system calls for all these testing commands.

Compared our tool with confine [27] and temporal [28], when generating the mapping table for the source-level standard library, the calling tree cannot be chained due to the function



(a) Library Function's Syscall (b) General Command's Functions

Fig. 5. Required syscall and library functions.



(a) General Command's Syscalls (b) Mitigation Syscall-related CVEs

Fig. 6. Effectiveness of TAILOR.

aliases. Through our analysis, all these aliases are caused by macro replacement and there are five replacement situations, i.e., weak symbol, strong symbol, versioned symbol, jumping table, and macro function. However, these two works cannot identify the jumping table and macro functions, leading to an incomplete mapping table. Thus, when testing our dataset using these two tools, most of them cannot work normally.

7.4. Preventing software vulnerabilities

From the Common Vulnerabilities and Exposures (CVE) dataset [29], we collect all vulnerabilities that are exploited via system calls in recent five years, 70 in total. We first introduce some vulnerability examples that can be prevented by blocking the related system calls.

CVE-2019-5489 [30]. This vulnerability exploits the `mincore` system call in Linux kernel 4.19. It allows an attacker to obtain page cache of other processes in the operating system and thus can leak sensitive information. Particularly, this vulnerability can be remotely exploited even if the attacker is in a remote server. However, a malicious application cannot exploit the vulnerability if we block the `mincore` system call.

CVE-2019-3901 [31]. This vulnerability exploits the `perf_event_open` system call. It can be reproduced in almost all Linux operating systems before version 4.8. The vulnerability is caused due to a race condition in `perf_event_open`. This race condition allows a malicious application to retrieve sensitive information. We can block the `perf_event_open` system call of an application if it does not need this system call.

CVE-2019-13648 [32]. This vulnerability exploits the `sigreturn` system call in Linux kernel 5.2.1. By constructing a specific frame, a local user can initiate an attack to make the system crash. However, in most platforms, this system call has been replaced by `rt_sigreturn`, meaning that most programs will not invoke `sigreturn`. If we can block this system call for an application, this vulnerability will be defended automatically.

CVE-2019-11503 [33]. This vulnerability can cause a restore permission bypass. When performing `chdir` system call to the current working directory, the `snap-confine` [34] can produce a race condition, if the version is less than 2.39. However, we can prevent an application from launching the attack by blocking the `chdir` system call.

About 75% system calls only correspond to 1 CVE, 19% system calls have been misused in 2 CVEs, and 7% system calls have appeared in 3 CVEs. Therefore, by extracting and restricting the necessary system calls for applications, we can prevent possible vulnerabilities from being exploited by commands. Fig. 6(b) shows how many vulnerabilities will be blocked if we leverage `seccomp` to restrict the system calls for a command based on its system call whitelist. We can see that at least 74% of vulnerabilities cannot be exploited by all commands and more than 90% vulnerabilities cannot be exploited for half of the commands. The results show that syscall related vulnerabilities can be effectively blocked by restricting available system calls for commands with TAILOR.

8. Related work

Security with System Calls A number of studies [35–39] have been proposed to enhance security with system calls. Ming et al. [35] check system call sliced segment equivalence to distinguish similarities between two execution flows. Yamauchi et al. [36] monitor privilege information change in the processing of system calls to detect escalation attacks. Moreover, other studies [37–39] develop advanced approaches by checking the patterns of invoking systems calls to detect malware. Different from the existing

studies, our work conducts fine-grained system call profiling for a given program. By restricting available system calls according to profiling results, we can significantly reduce the attack surface of both programs and systems.

As far as we know, our work presents the first tool to automatically profile fine-grained system calls for programs. One study close to our work is Speaker [10], which profiles and dynamically updates available system calls for container applications to reduce the attack surface. However, Speaker requires users to manually run programs to find their system calls. It is time-consuming and difficult to traverse all branches of different programs to capture system calls that may be invoked. Compared to Speaker, TAILOR can efficiently and automatically profile complete system calls for programs. Therefore, TAILOR guarantees that programs can work normally while reducing the attack surface.

Also, our tool builds a mapping table for the standard library, which most applications are used to invoke system calls. Thus it can be expanded to different scenes, for example, in computing scenes [40] to enhance the Docker container security, in feature selection scenes [41] to guarantee algorithm security, in multimedia scenes [42,43] to protect communication sides, and in image processing scenes [44] to prevent model attacking. In one word, TAILOR is basic infrastructure, on which we can build plenty of implementations.

Static Analysis. The static analysis technique [45] has been widely used in various aspects of analyzing programs. A number of studies [46–49] use static analysis to generate data flow and control flow, which can be used to detect bugs and vulnerabilities [46,49–51], check the integrity to prevent privilege escalation [47,50,51] and reduce unnecessary path access in programs [48]. Some studies also use symbolic execution to statically analyze programs for finding bugs [52] and validating patches [53]. Moreover, there are studies using static analysis to build FCG for programs [54] and leverage FCGs to identify malware [55–57]. However, none of the studies have built FCGs for library functions to accurately find the mapping between library functions and system calls.

9. Conclusion

In this paper, we develop a toolkit named TAILOR to generate the required system call whitelist to reduce the attack surface for any specific user process. TAILOR is capable of mapping library functions to their corresponding system calls and obtaining the invoked system calls via the standard library for any application. TAILOR solves the problems during the source-level standard library analysis, i.e., the difficulty in macro function identification, the unconnected calling tree caused by function replacement, and the difficulty in identifying the function scope via static analysis. We use TAILOR to study the glibc library and the experimental results show that the library functions of glibc invoke at least 11 system calls and at most 39 system calls. Our evaluation of 50 general commands shows that TAILOR can dramatically reduce the number of available system calls in the whitelist and thus effectively defeat a number of system calls related CVE vulnerabilities. In future research, we will divide the running stage of an application into different stages and customize the seccomp rule for each stage, which can implement a fine-grained security measure.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Table 1
General commands and required system calls. Bin. is short for binary, Func. is short for function and Sys. is short for system calls.

Bin.	Func.	Sys.	Bin.	Func.	Sys.	Bin.	Func.	Sys.
id	66	48	dash	91	63	bzcat	44	42
cp	139	63	date	80	38	zdump	40	36
rm	77	45	diff	121	45	base64	61	36
dd	84	42	dpkg	142	75	bitmap	23	35
xz	115	46	echo	49	35	catman	43	37
dir	114	47	find	160	57	zipinfo	87	48
cat	58	37	grep	119	40	dirname	50	35
zip	100	48	host	39	36	apt-get	122	36
sed	122	49	kill	69	42	calender	80	49
top	135	52	nice	55	38	apt-mark	93	36
cmp	68	36	bzip2	43	42	basename	51	35
cut	66	36	paste	56	36	apt-cache	53	36
man	83	45	chacl	35	35	ausyscall	11	35
yes	50	35	chgrp	77	47	apt-cdrom	65	37
arch	54	37	chmod	69	39	apt-config	53	36
comm	59	36	chown	79	47	apt-sortpkgs	53	36
curl	127	48	clear	29	36			

Acknowledgments

We would like to thank our anonymous reviewers for their valuable comments and suggestions. We would also like to thank Pengbin Feng and Zeyu Zhang for their feedback and advice. This work is supported by U.S. National Science Foundation, Division of Computer and Network Systems under Grant No. 1815650, National Natural Science Foundation of China under Grant No. 61272452, and Hubei Province Key Research and Development Program of China under Grant No. 2020BAA003.

Appendix

See Table 1.

References

[1] User space and kernel space, 2022, https://en.wikipedia.org/wiki/User_space_and_kernel_space.
[2] A. Reina, A. Fattori, L. Cavallaro, A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors, EuroSec. (2013).
[3] D. Canali, A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, E. Kirda, A quantitative study of accuracy in system call-based malware detection, in: Proceedings of the 2012 International Symposium on Software Testing and Analysis, ACM, 2012, pp. 122–132.
[4] Y. Shao, J. Ott, Y.J. Jia, Z. Qian, Z.M. Mao, The misuse of android unix domain sockets and security implications, in: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, ACM, 2016, pp. 80–91.
[5] SECCOMP Man, 2022, http://man7.org/linux/man-pages/man2/seccomp.2.html.
[6] J. Winter, Trusted computing building blocks for embedded linux-based ARM trustzone platforms, in: Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing, ACM, 2008, pp. 21–30.
[7] T. Kim, N. Zeldovich, Practical and effective sandboxing for non-root users, in: Presented As Part of the 2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13), 2013, pp. 139–144.
[8] TwistLock Inc., 2022, https://www.twistlock.com.
[9] Aqua security software ltd., 2022, https://www.aquasec.com.
[10] L. Lei, J. Sun, K. Sun, C. Shenefiel, R. Ma, Y. Wang, Q. Li, SPEAKER: SPlit-phase execution of application containers, in: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Springer, 2017, pp. 230–251.
[11] M. Bagherzadeh, N. Kahani, C.-P. Bezemer, A.E. Hassan, J. Dingel, J.R. Cordy, Analyzing a decade of linux system calls, in: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), IEEE, 2018, pp. 267–267.
[12] J.-B. Yunès, Difference between write and printf, 2022, https://stackoverflow.com/questions/21084218/difference-between-write-and-printf.
[13] alpha9eek, What is the difference between a library call and a system call in linux?, 2022, https://unix.stackexchange.com/questions/6931/what-is-the-difference-between-a-library-call-and-a-system-call-in-linux.

[14] Why system calls are limited to c language as far as I see, 2022, https://softwareengineering.stackexchange.com/questions/343783/why-system-calls-are-limited-to-c-language-as-far-as-i-see.
[15] J. Edge, A seccomp overview, 2022, https://lwn.net/Articles/656307/.
[16] CVE-2020-12659 Detail, 2020, https://nvd.nist.gov/vuln/detail/CVE-2020-12659.
[17] CVE-2020-12654 Detail, 2020, https://nvd.nist.gov/vuln/detail/CVE-2020-12654.
[18] CVE-2018-10924 Detail, 2018, https://www.cvedetails.com/cve/CVE-2018-10924.
[19] CVE-2017-14140 Detail, 2017, https://www.cvedetails.com/cve/CVE-2017-14140.
[20] CVE-2016-8440 Detail, 2016, https://www.cvedetails.com/cve/CVE-2016-8440.
[21] CVE-2016-1883 Detail, 2016, https://www.cvedetails.com/cve/CVE-2016-1883.
[22] Clang, 2022, https://clang.llvm.org/index.html.
[23] The linux man-pages project, 2022, https://www.kernel.org/doc/man-pages/.
[24] Strace, 2022, https://en.wikipedia.org/wiki/Strace.
[25] GNU Cflow, 2022, https://www.gnu.org/software/cflow/.
[26] Readelf, 2022, https://linux.die.net/man/1/readelf.
[27] S. Ghavamnia, T. Palit, A. Benameur, M. Polychronakis, Confine: Automated system call policy generation for container attack surface reduction, in: 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020), USENIX Association, San Sebastian, 2020, pp. 443–458.
[28] S. Ghavamnia, T. Palit, S. Mishra, M. Polychronakis, Temporal system call specialization for attack surface reduction, in: 29th USENIX Security Symposium (USENIX Security 20), USENIX Association, 2020, pp. 1749–1766.
[29] Common vulnerabilities and exposures, 2022, https://cve.mitre.org/.
[30] CVE-2019-5489 Detail, 2019, https://nvd.nist.gov/vuln/detail/CVE-2019-5489.
[31] CVE-2019-3901 Detail, 2019, https://nvd.nist.gov/vuln/detail/CVE-2019-3901.
[32] CVE-2019-13648 Detail, 2019, https://nvd.nist.gov/vuln/detail/CVE-2019-13648.
[33] CVE-2019-11503 Detail, 2019, https://nvd.nist.gov/vuln/detail/CVE-2019-11503.
[34] snapd, 2022, https://github.com/snapcore/snapd.
[35] J. Ming, D. Xu, Y. Jiang, D. Wu, Binsim: Trace-based semantic binary diffing via system call sliced segment equivalence checking, in: 26th {USENIX} Security Symposium ({USENIX} Security 17), 2017, pp. 253–270.
[36] T. Yamauchi, Y. Akao, R. Yoshitani, Y. Nakamura, M. Hashimoto, Additional kernel observer to prevent privilege escalation attacks by focusing on system call privilege changes, in: 2018 IEEE Conference on Dependable and Secure Computing (DSC), IEEE, 2018, pp. 1–8.
[37] B. Kolosnjaji, A. Zarras, G. Webster, C. Eckert, Deep learning for classification of malware system call sequences, in: Australasian Joint Conference on Artificial Intelligence, Springer, 2016, pp. 137–149.
[38] V. Radhakrishna, P.V. Kumar, V. Janaki, A novel similar temporal system call pattern mining for efficient intrusion detection., J. UCS 22 (4) (2016) 475–493.
[39] R. Canzanese, S. Mancoridis, M. Kam, System call-based detection of malicious processes, in: 2015 IEEE International Conference on Software Quality, Reliability and Security, IEEE, 2015, pp. 119–124.
[40] C. Stergiou, K.E. Psannis, B.B. Gupta, Y. Ishibashi, Security, privacy & efficiency of sustainable cloud computing for big data & IoT, Sustainable Computing: Informatics and Systems 19 (2018) 174–184.
[41] M. Alweshah, S. Al Khalaileh, B.B. Gupta, A. Almomani, A.I. Hammouri, M.A. Al-Betar, The monarch butterfly optimization algorithm for solving feature selection problems, Neural Comput. Appl. (2020) 1–15.
[42] H. Wang, Z. Li, Y. Li, B. Gupta, C. Choi, Visual saliency guided complex image retrieval, Pattern Recognit. Lett. 130 (2020) 64–72.
[43] M.S. Hossain, G. Muhammad, W. Abdul, B. Song, B.B. Gupta, Cloud-assisted secure video transmission and sharing framework for smart cities, Future Gener. Comput. Syst. 83 (2018) 596–606.
[44] S. AlZu'bi, M. Shehab, M. Al-Ayyoub, Y. Jararweh, B. Gupta, Parallel implementation for 3d medical volume fuzzy segmentation, Pattern Recognit. Lett. 130 (2020) 312–318.
[45] Static analysis, 2022, https://en.wikipedia.org/wiki/Static_pro-gram_analysis.
[46] Y. Sui, J. Xue, Svf: interprocedural static value-flow analysis in LLVM, in: Proceedings of the 25th International Conference on Compiler Construction, ACM, 2016, pp. 265–266.
[47] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, W. Lee, Enforcing kernel security invariants with data flow integrity., in: NDSS, 2016.
[48] J. Lerch, J. Späth, E. Bodden, M. Mezini, Access-path abstraction: Scaling field-sensitive data-flow analysis with unbounded access paths (t), in: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2015, pp. 619–629.
[49] N. Carlini, A. Barresi, M. Payer, D. Wagner, T.R. Gross, Control-flow bending: On the effectiveness of control-flow integrity, in: 24th {USENIX} Security Symposium ({USENIX} Security 15), 2015, pp. 161–176.
[50] A.J. Mashtizadeh, A. Bittau, D. Boneh, D. Mazières, Ccfi: Cryptographically enforced control flow integrity, in: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, ACM, 2015, pp. 941–951.
[51] B. Niu, G. Tan, Per-input control-flow integrity, in: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, ACM, 2015, pp. 914–926.
[52] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, G. Vigna, Driller: Augmenting fuzzing through selective symbolic execution., in: NDSS, 16, (2016) 2016, pp. 1–16.
[53] D.A. Ramos, D. Engler, Under-constrained symbolic execution: Correctness checking for real code, in: 24th {USENIX} Security Symposium ({USENIX} Security 15), 2015, pp. 49–64.
[54] D. Grove, C. Chambers, A framework for call graph construction algorithms, ACM Transactions on Programming Languages and Systems (TOPLAS) 23 (6) (2001) 685–746.
[55] S. Shang, N. Zheng, J. Xu, M. Xu, H. Zhang, Detecting malware variants via function-call graph similarity, in: 2010 5th International Conference on Malicious and Unwanted Software, IEEE, 2010, pp. 113–120.
[56] X. Hu, T.-c. Chiueh, K.G. Shin, Large-scale malware indexing using function-call graphs, in: Proceedings of the 16th ACM Conference on Computer and Communications Security, ACM, 2009, pp. 611–620.
[57] M. Hassen, P.K. Chan, Scalable function call graph-based malware classification, in: Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, ACM, 2017, pp. 239–248.