Fine-grained Program Partitioning for Security

Zhen Huang
zhen.huang@depaul.edu
DePaul University
Chicago, Illinois, USA

Trent Jaeger
tjaeger@psu.edu
Pennsylvania State University
State College, Pennsylvania, USA

Gang Tan
gtan@psu.edu
Pennsylvania State University
State College, Pennsylvania, USA

ABSTRACT

Complex software systems are often not designed with the principle of least privilege, which requires each component be given the minimum amount of privileges to function. As a result, software vulnerabilities in less privileged code can lead to privilege escalation, defeating security and privacy. Privilege separation is the process of automatically partitioning a software system into least privileged components, and we argue that it is effective at reducing the attack surface. However, previous privilege-separation systems do not provide fine-grained separation of privileged code and non-privileged code co-existing in the same function for C/C++ applications. We propose a fine-grained partitioning technique for supporting fine-grained separation in automatic program partitioning. The technique has been applied to a set of security-sensitive networking and interactive programs. Results show that it can automatically generate executable partitions for C applications; further, partitioned programs incur acceptable runtime overheads.

CCS CONCEPTS

• **Security and privacy → Software and application security**.

KEYWORDS

software security, program partitioning, principle of least privilege, program analysis

ACM Reference Format:
Zhen Huang, Trent Jaeger, and Gang Tan. 2021. Fine-grained Program Partitioning for Security. In *14th European Workshop on System Security (EuroSec' 21)*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3447852.3458717

1 INTRODUCTION

Software vulnerabilities remain a critical issue of computer security, despite decades of research on identifying, preventing, and fixing them. Although vulnerabilities often exist in non-privileged code, exploiting such vulnerabilities can allow adversaries to gain access to privileged code that co-exists with non-privileged code in the same program. For example, such vulnerabilities were discovered recently in both Linux and Windows applications [1, 3].

EuroSec '21, April 26, 2021, Edinburgh, Scotland, UK
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8337-0/21/04.
https://doi.org/10.1145/3447852.3458717

An effective approach to preventing such exploits from accessing privileged code is to separate non-privileged code from privileged code in a program. Privileged code should be made to reside in its own partition and be isolated from a partition that hosts non-privileged code so that the compromise of the non-privileged partition does not directly lead to the compromise of the privileged partition. The two partitions communicate via a carefully designed, controlled interface. This design follows the principle of least privilege and significantly raises the bar for adversaries to gain privilege escalation by exploiting vulnerabilities in non-privileged code.

To aid developers to partition a program, several automatic program-partitioning tools [6, 7, 14, 18] have been proposed to privilege separate C/C++ applications, from a small number of user annotations about how to identify privileged operations. For instance, Privtrans [7] requires users to annotate sensitive data and any operations that access sensitive data are treated as privileged operations. These systems demonstrate that automatic program partitioning can be practical.

However, existing tools suffer from a major drawback that prevent them from being widely adopted. Most tools partition a program at the granularity of functions. They partition the program's code into two sets of functions, but do not split individual functions. This is mainly because programs are already structured using functions as the basic unit; when a function calls another function in a different partition, the call naturally becomes a Remote Procedure Call (RPC) after partitions are isolated. However, there are cases when a split should occur within functions. For example, a function can call both process management functions, which may have to be executed in a non-sensitive partition, and privileged functions, which must be executed in a sensitive partition. Therefore, an ideal partitioning tool should support fine-grained partitioning, which can split such individual functions.

In this paper, we describe a technique that makes progress toward addressing the aforementioned weakness. This technique has been implemented in a prototype system. Our major contributions are as follows:

- The technique supports fine-grained partitioning that partitions individual functions of programs; it addresses the challenge to separate privileged code and non-privileged code co-existing in the same function.
- We implemented the techniques in a prototype. Our evaluation on the prototype shows that our fine-grained partitioning is effective and efficient. The prototype is open-sourced at https://github.com/huang-zhen/ProgramPartitioning.

2 FINE-GRAINED PARTITIONING

```
void load_identity_file(Identity *id) {
  char *prompt = "Enter␣passphrase␣for␣key:␣";
  char *passphrase = read_passphrase(prompt, 0);
  int ret = sshkey_load_private_type(id->
    filename, passphrase);
  if (ret != 0)
    error("Load␣key␣failed");
  return ret;
}
```

Figure 1: A function adapted from *openssh*. It illustrates the co-existence of calls to both a privileged function `sshkey_load_private_type` and an NPS function `read_passphrase` within the function.

```
void update_gecos(char *user, char *gecos){
  const struct passwd *pw;
  struct passwd newpw;

  if (pw_open(O_RDWR, cl) == 0) {
    fprintf(stderr,"cannot␣open␣file\n");
    fail_exit(E_NOPERM);
  }
  pw = pw_locate(user);
  if (NULL == pw) {
    fprintf(stderr,"user␣does␣not␣exist\n"),
    fail_exit(E_NOPERM);
  }
  newpw = *pw;
  newpw.pw_gecos = gecos;
  if (pw_update(&newpw) == 0) {
    fprintf(stderr,"failed␣to␣prepare␣the␣new␣
    entry'\n");
    fail_exit(E_NOPERM);
  }
  if (pw_close() == 0) {
    fprintf(stderr,"failure␣while␣writing␣
    changes\n");
    SYSLOG("failure␣while␣writing␣changes");
    fail_exit(E_NOPERM);
  }
}
```

Figure 2: A function adapted from *chfn* in the Linux shadow-utils package. It illustrates the co-existence of calls to privileged functions such as `pw_open` and `pw_update`, and NPS functions such as `fprintf` and `fail_exit`.

Function-level program partitioning is sufficient for many usage scenarios as demonstrated by prior work [7, 8, 12–16, 18–20]. However, it does not support the scenarios when a function contains both calls to privileged functions that must stay in the sensitive partition and calls to non-privileged functions that must stay outside.

We refer to program partitioning at a granularity finer than functions as *fine-grained program partitioning*. As an initial step, we present a special form of fine-grained partitioning to address two common patterns we identified in a set of networking programs and interactive programs.

In the first pattern, a user-interactive function is called to gather information from the user and the information is then used to call a privileged function. The first pattern is illustrated in Figure 1, in which `load_identity_file` calls `read_passphrase`, a user-interactive function, and `sshkey_load_private_type`, a privileged function. We call this pattern *non-privileged-to-privileged* pattern.

In the second pattern, a privileged function is called first and the result of the call determines whether to execute of one or more non-privileged functions. As illustrated in Figure 2, `update_gecos` calls several privileged functions such as `pw_open` and `pw_locate`, and non-privileged functions such as `fprintf` and `fail_exit`. We call this pattern *privileged-to-non-privileged* pattern.

In both patterns, a function contains both calls to privileged functions and calls to non-privileged functions. We call the latter non-privileged-partition specific (NPS) functions. A function is an NPS function for reasons as follows: (1) sandboxing restrictions dictate that certain library functions must be executed outside privileged domains; e.g., an SGX sandbox cannot directly issue OS syscalls and has to transition outside of the sandbox to call many C library functions that request OS services; (2) file I/O functions such as the `fprintf` family of functions should stay in a non-privileged partition; (3) process management functions such as the one for process termination should stay in a non-privileged partition. One key reason why functions of categories (2) and (3) should stay in a non-privileged partition is because of our design of using OS processes as the isolation mechanism. A program is split in a way so that the main functionality including I/O stay in a non-privileged process, which issues calls to a privileged process hosting privileged functions. In this design, the non-privileged process performs I/O and process management (e.g., starting and terminating the privileged process).

To illustrate the challenge to separate privileged code and non-privileged code co-existing in the same function, Figure 1 shows the `load_identity_file` function that loads a private key file. It calls an NPS function `read_passphrase` to prompt the user to enter the passphrase for the private key file and then calls a privileged function `sshkey_load_private_type` to read the private key file into memory and check whether the passphrase is correct or not. Figure 2 shows the `update_gecos` function that updates sensitive information in the password file. It calls privileged function `pw_lock` to lock the password file and privileged function `pw_update` to update the information stored in the password file. If any failure occurs during the call to a privileged function, `update_gecos` calls functions `fprintf` and `fail_exit` to output an error message and terminate the program execution. We treat `fprintf` and `fail_exit` as NPS functions for reasons we discussed earlier.

To understand the coding conventions of this kind of functions, we studied the source code of networking programs including *openssh* and *wget*, and interactive programs from the Linux shadow-utils package. We find that there are two typical coding conventions involving intertwined calls to privileged functions and NPS functions. First, a function can call I/O functions to let the user enter some information and then call privileged functions using the entered information. Second, a function can call privileged functions and then call I/O functions to display error messages and call process management functions to terminate program execution, based on the return values of the calls to the privileged functions.

There may exist other coding conventions that involve intertwined calls to privileged functions and NPS functions. For example, calls to NPS functions may not be control dependent on calls to privileged functions. For this work, we focus on the coding convention representing the non-privileged-to-privileged pattern and the privileged-to-non-privileged pattern. We call a function using the coding convention a *hotspot function*.

A naive solution is to put a hot spot function in the non-sensitive partition and change each of its privileged call to a remote call to the sensitive partition. For example, all calls to privileged functions such as pw_lock will be replaced with inter-partition calls. But this would result in multiple inter-partition calls and thus incur the runtime overhead of multiple rounds of cross-partition communication. In Section 5.2, we will show that the runtime overhead of partitioning is approximately proportional to the number of inter-partition calls.

To reduce the performance overhead, our solution splits a hotspot function into two functions: one *primary function* that contains the calls to privileged functions but not NPS functions and one *secondary function* that contains the calls to NPS functions. The primary function is executed in the privileged partition, while the secondary function is executed in the non-privileged partition. Unlike the naive solution, this solution requires only one inter-partition call from the secondary function to the primary function.

The primary function is a clone of the original function with the code change that replaces a group of consecutive calls to NPS functions with a return statement. For example, line 6 and line 7 in Figure 2 are executed when the call to privileged function pw_open fails; so they are replaced with a return statement that returns an error return value -1. Similarly, since line 17 and line 18 are executed when the call to pw_update fails, they are replaced with a return statement that returns an error return value -2.

The secondary function consists of a call to the primary function and a switch statement that checks the return value from the primary function and invokes calls to NPS functions corresponding to the return value. Figure 3 shows the secondary function created for the function shown in Figure 2. As we can see, the secondary function first calls the primary function. If the call returns -1, it indicates a failure in the call to pw_lock in the primary function; so the secondary function performs the work of lines 6 and 7 in Figure 2. Other cases are similar. As a result, the primary function and the secondary function work together to retain the original functionality of update_gecos.

The creation of the primary function and the secondary function is composed of four steps: 1) identifying hot spot functions that need to be split; 2) creating the primary function for each identified function; 3) creating the secondary function for each identified function; 4) substituting the calls to identified functions to the calls to their corresponding secondary functions.

```
1  void update_gecos_sec(char *user, char *gecos){
2    int *ret = update_gecos_pri(user,gecos);
3    switch (*ret) {
4    case -1: /* pw_open */
5      fprintf(stderr,"cannot␣open␣file\n");
6      fail_exit(E_NOPERM);
7      break;
8    case -2: /* pw_locate */
9      fprintf(stderr, "user␣does␣not␣exist\n");
10     break;
11   ...
12   ...
13   }
14 }
```

Figure 3: The secondary function created from function update_gecos. Function update_gecos_pri is the primary function created from function update_gecos.

Algorithm 1 Identifying hot spots in a function

Input: G is the set of all function calls in a function F; PL is the list of privileged functions; NPL is the list of NPS functions
Output: H is the set of hot spots in F

```
H ← ∅
for function call FC ∈ G do
    if (Callee(FC) ∈ PL) then
        for statement C ∈ CheckCallRet(FC) do
            for branch B ∈ GetBranches(C) do
                for statement S ∈ B do
                    if (IsCall(S) ∧ Callee(S) ∈ NPL) then
                        H ← H ∪ FC
```

2.1 Identifying Hot Spot Functions

For the privileged-to-non-privileged pattern, we define a *hot spot* as a call to privileged function on which a sequence of one or more calls to an NPS function are control-dependent. The sequence must end with either an NPS call that would terminate the program execution or a return statement. We use Algorithm 1 to identify the privileged-to-non-privileged pattern.

In the algorithm, function Callee returns the callee of a statement if the statement is a function call. Function CheckCallRet returns a set of conditional statements that checks the return value of a function call. Function GetBranches returns a set of branches for a conditional statement. And function IsCall returns true if a statement is a function call or false otherwise.

2.2 Creating Primary Functions

We create the primary function for each hot spot function in four steps. First, we clone the code of the hot spot function as the initial code of the primary function. Second, we modify the prototype of the function and the original return statements in the function if needed. If the return type of the function is not void, we add an additional pointer parameter as the original return value and replace

Algorithm 2 Creating a primary function

Input: F is the set of all statements in a hot spot function of a program; H is the set of hot spot statements in F; B is the set of all statements in a branch of an `if` or `switch` statement

Output: G is the set of all statements in the primary function corresponding to F; M is the set of matching pairs of hot spot return values and NPS call statements

```
$G \leftarrow F$
$M \leftarrow \emptyset$
$ReturnValue \leftarrow 0$
for statement $S \in G$ do
    if ($S \in H$) then
        for branch $B \in$ GetBranches($S$) do
            $ReturnValue \leftarrow ReturnValue - 1$
            for NPS call $C \in B$ do
                $G \leftarrow G - C$
                $M \leftarrow M \cup (ReturnValue, C)$
            $R \leftarrow$ CreateReturnStatement($ReturnValue$)
            $G \leftarrow G \cup R$
```

the original return statements as assignments to the additional pointer parameter. Then we change the return type of the function to `int`. Third, we label each branch of all hot spots within the code with a unique return value, i.e. a hot spot return value. Last, we remove NPS function calls on a branch of a hot spot and add a return statement at the end of the branch that returns the hot spot return value corresponding to the branch. To aid the creation of the secondary function, this step also produces a mapping from a hot spot return value to the corresponding sequence of NPS function calls.

Algorithm 2 shows how we create the primary function for a hot spot function. Function `GetBranches` returns a set of branches of an `if` or a `switch` statement. Function `CreateReturnStatement` creates a return statement that returns a specified value.

2.3 Creating Secondary Functions

In the secondary function, we create a call to the primary function, followed by a `switch` statement. Based on the mapping between each hot spot return value of the primary function and its corresponding NPS function calls, which is produced in the creation of the primary function, each case of the `switch` statement calls NPS functions based on the hot spot return value returned from the call to the primary function. If the original hot spot function has return values, we add return statements to return the value passed via the additional return value parameter of the primary function.

3 COMMUNICATION BETWEEN PARTITIONS

To work together as a single program, partitions need to communicate with one another. In our design, each partition of a partitioned program is packaged as a standalone program and loaded into a separate process. Each partition uses Remote Procedure Call (RPC) to communicate with another partition. Hence, privileged code that needs to be invoked from the insensitive partition is exposed to the insensitive partition as one or more RPC functions.

As discussed in Section 2, a primary function and a secondary function are produced for each hot spot function. The primary function is implemented as an RPC function running in the sensitive partition and the original hot spot function is replaced by the secondary function that invokes the corresponding primary function to perform privileged operations.

4 IMPLEMENTATION

We have implemented our technique in a prototype on Linux. It works on C/C++ programs and uses Talos [10] to get various information on the source code of the programs that are needed by our algorithms.

For each target program, it creates the source code of an RPC server program that represents the sensitive partition and modifies the target program to invoke functions in the RPC server program to perform privileged operations.

To allow the RPC server program to perform privileged operations, the prototype sets the `setuid` bit on the RPC server program.

5 EVALUATION

We evaluate the effectiveness and the performance overhead of our fine-grained program partitioning. Our evaluation is conducted on a set of networking programs and interactive programs on a workstation running 64-bit Ubuntu 16.04 with an Intel Core i7-7700 3.6GHz CPU and 16GB of RAM.

5.1 Effectiveness

Networking programs. `ssh` is an utility included in `OpenSSH` [4], which implements the client-side of the SSH protocol. We consider the RSA private key loaded by ssh as the sensitive data. To measure performance overhead, we used the partitioned `ssh` to log into an SSH server running on the same workstation.

`wget` is a program for downloading files from an HTTP or FTP server [2]. We treat the downloaded file as the sensitive data because malicious content may be embedded in the file. To measure performance overhead, we used the partitioned `wget` to download a 1KB file from an FTP server running on the same workstation.

Interactive programs. We also experimented on a set of interactive programs from the Linux shadow-utils package [5]. There are over 30 programs in this package. Many of them do not access security-sensitive information; for example, program "groups" just prints a user's group information. Some of the programs are difficult to set up to run and experiment with; for example, "login" starts a login session. So we excluded those programs. The programs we included are presented in Table 1.

For interactive programs, we discuss only how `chsh` is privilege separated; other programs are similar. `chsh` is a setuid program that enables a user to change his/her login shell. Any regular Linux user can invoke it and, because of setuid, it runs with root privileges. It works by updating the user's corresponding shell entry in `/etc/passwd` with the new shell supplied by the user. To privilege separate `chsh`, we annotated variables `pw` and `pwent` as sensitive; they hold the old and new entry in the password file, respectively. Function `update_shell()` is then identified as the only privileged function. Then `chsh` is separated into an executable with `update_shell()`, called `pchsh`, and an executable with the

Table 1: Partitioning results of benchmark programs.

Benchmark	SLOC	Sensitive Data	# of functions/ privileged funcs
openssh	99,507	private key file	975/4
wget	86,294	downloaded file	730/3
chfn	928	password	12/3
chsh	763	password	10/1
chage	1,314	password	15/3
passwd	1,581	password, shadow	14/5
gpasswd	1,524	group	21/3
useradd	3,092	password, group, shadow	25/5
userdel	2,048	password, group, shadow	12/4
pwconv	667	password, group, shadow	5/1

rest of the code, still called chsh. After separation, pchsh has the setuid bit on, while the new chsh does not. This way, only a small amount of code (i.e., those in pchsh) is run with higher privileges, reducing the attack surface of chsh.

5.2 Performance

Overhead of RPC calls. To understand the effect of fine-grained partitioning, we measured the runtime overhead of RPC calls and result is shown in Figure 4. The RPC calls are made between a client program and a server program running on the same computer so there is no networking overhead. A function in the client program makes a specific number of RPC calls to a remote function on the server. We vary the number of RPC calls made within the function and measure the execution time of the function for different number of RPC calls.

Because the execution time of an RPC call depends on not only the communication overhead but also the execution time of code of the remote function, we choose to use three types of basic functions for measurements. They include a *null function* that does not take any parameter and returns to its caller without any return value, an *integer function* that takes a single 32-bit integer parameter and returns twice the value of the parameter back to its caller, and a *string function* that takes a character string parameter and returns a constant 20-character string back to its caller.

The figure shows the relation between the number of RPC calls and the execution time for the three types of functions. For all functions, there is roughly a linear relation between the number of RPC calls and the execution time: the more RPC calls is invoked, the higher is the execution time. The null function and the integer function have approximately the same amount of execution time, while the string function has an execution time significantly longer than them. The result shows that we can reduce the runtime overhead by reducing the number of RPC calls.

Fine-grained partitioning. We compare our fine-grained partitioning with a naive partitioning described in Section 2. Instead of creating a primary function that embodies all the calls to privileged functions invoked from a single function, the naive partitioning turns each such call into a separate RPC call.

Table 2 shows the results of the comparison. Column "# Reduced RPC calls" presents the difference between the number of RPC calls occurred in the naive partitioning and the number of RPC calls

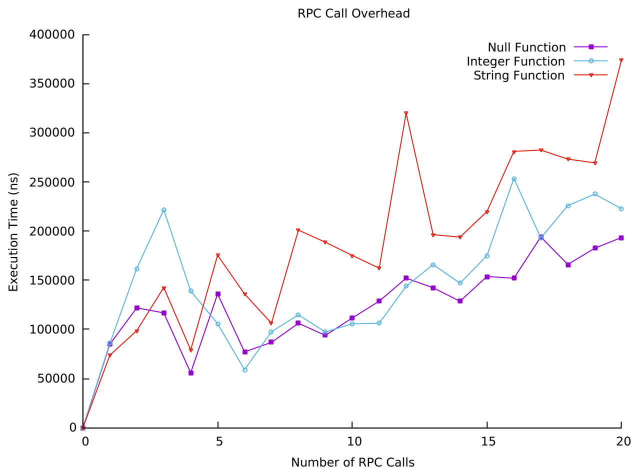

Figure 4: Overhead of RPC Calls

Table 2: Results of fine-grained partitioning.

Benchmark	# Hot spot functions	# Reduced RPC calls	Speedup
openssh	1	2	1.32x
wget	1	2	1.06x
chfn	1	6	1.10x
chsh	1	5	1.19x
chage	3	6	1.07x
passwd	2	9	1.03x
gpasswd	3	3	1.03x
useradd	5	12	1.01x
userdel	4	12	1.07x
pwconv	1	192	3.90x
AVERAGE	2.5	30.6	1.38x

occurred in our fine-grained partitioning. Column " Speedup" gives the result of dividing the program's execution time for the naive partitioning by the execution time for the fine-grained partitioning.

As we can see, both networking programs and interactive programs can benefit from fine-grained partitioning. These programs have between one to five hotspot functions. By applying fine-grained partitioning to these hotspot functions, as high as 192 RPC calls are eliminated. As a result, the runtime performance of these programs is improved by an average of 38%. In general, the more the number of reduced RPC calls is, the higher the performance improvement we can get from fine-grained partitioning.

The execution time of these programs under typical inputs and the performance overhead of the partitioned programs compared to their unpartitioned versions are shown in Table 3. On average, there is a 5.2% of performance overhead. The shadow-util programs are short-running programs; even after partitioning, the amount of total running time is quite acceptable.

6 RELATED WORK

Many program partitioning tools have been proposed [6–9, 11–20]. Based on the extent of support they provide to programmers, they

Table 3: Runtime overhead imposed by partitioning.

Benchmark	Task	Runtime (ms)	Overhead (%)
openssh	login into an SSH server	3.0	8.9%
wget	download a file from an FTP server	54.5	7.4%
chfn	change real user info	66.7	-5.8%
chsh	change user shell	63.9	-1%
chage	change password expiry info	58.5	2.4%
passwd	change user password	67.9	10%
gpasswd	chage group password	58.7	4.4%
useradd	add users	177.4	3.5%
userdel	delete users	176.9	1.4%
pwconv	create shadow from password	0.4	76%
GEOMEAN		34.0	5.2%

can be roughly categorized into tools that aid program partitioning and tools that automatically partition programs.

Some tools aid programmers in partitioning programs manually. Wedge dynamically collects statistics about how memory regions are used in a program [6] to help programmers identify partitioning boundaries. Privman provides a set of library functions that can be used by programmers to ease the task of partitioning programs [11].

Automatic program partitioning splits a program into two or more partitions with minimum programmer interference. Several tools work on C/C++ programs. Using static analysis, Privtrans and PtrSplit partition a C program into a non-privileged partition and a privileged partition, which communicate via RPC [7, 13]. Unlike Privtrans and PtrSplit, ProgramCutter employs dynamic profiling information to partition programs [18]. It aims to achieve a balance between performance and security in program partitioning by using graph partitioning on a dynamic dependency graph built from the profiling information. Combining static analysis and dynamic analysis, SeCage decomposes a program into multiple partitions, each of which can have its own sensitive data [14].

For Java programs, Jif/split performs automatic partitioning based on security annotations and a user-specified trust relationship between partitions [19, 20]. Swift extends the approach used by Jif/split to web applications that involve more complex data structures and control flows [8].

There are also program partitioning tools that take advantage of the hardware supported Trusted Execution Environments [12, 16].

7 LIMITATIONS

Our fine-grained program partitioning focuses on two coding patterns involved in individual functions that have intertwined privileged code and non-privileged code: privileged-to-non-privileged and non-privileged-to-privileged. There exist other patterns. We plan to address those patterns in our future work.

The prototype is not completely automated. For a target program, the prototype can automatically produce the RPC server program, representing the sensitive partition, and makes the necessary modifications directly on the target program so that the modified target program represents the insensitive partition. But it relies on developers to create the RPC specification file needed by both the RPC server program and the modified target program. To aid the developers, it provides the developers the list of RPC functions and the parameters for each of the RPC functions.

8 CONCLUSION

Automatically privilege separating an application has been effective in improving software systems' security, especially when those systems are written in memory-unsafe languages. In this paper, we describe fine-grained program partitioning that supports separation of intertwined privileged code and non-privileged code within functions. Our evaluation indicates that fine-grained program partitioning is effective and improves performance in partitioned programs.

REFERENCES

[1] GNU Beep 1.3 - 'HoleyBeep' Local Privilege Escalation. https://www.exploit-db.com/exploits/44452/.

[2] GNU Wget. https://www.gnu.org/software/wget/.

[3] Microsoft Office CVE-2018-8412 Privilege Escalation Vulnerability. https://www.symantec.com/security-center/vulnerabilities/writeup/105014.

[4] OpenSSH. https://www.openssh.com/.

[5] Shadow Utils. https://www.centos.org/docs/5/html/5.5/technical-notes/shadow-utils.html.

[6] Bittau, A., Marchenko, P., Handley, M., and Karp, B. Wedge: splitting applications into reduced-privilege compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (2008), pp. 309–322.

[7] Brumley, D., and Song, D. Privtrans: Automatically partitioning programs for privilege separation. In *13th Usenix Security Symposium* (2004), pp. 57–72.

[8] Chong, S., Liu, J., Myers, A., Qi, X., Vikram, K., Zheng, L., and Zheng, X. Secure web applications via automatic partitioning. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)* (Oct. 2007), pp. 31–44.

[9] Clements, A. A., Almakhdhub, N. S., Bagchi, S., and Payer, M. ACES: Automatic compartments for embedded systems. In *27th USENIX Security Symposium (USENIX Security 18)* (Baltimore, MD, Aug. 2018), USENIX Association, pp. 65–82.

[10] Huang, Z., D'Angelo, M., Miyani, D., and Lie, D. Talos: Neutralizing vulnerabilities with security workarounds for rapid response. In *2016 IEEE Symposium on Security and Privacy (SP)* (May 2016), pp. 618–635.

[11] Kilpatrick, D. Privman: A library for partitioning applications. In *USENIX Annual Technical Conference, FREENIX track* (2003), pp. 273–284.

[12] Lind, J., Priebe, C., Muthukumaran, D., O'Keeffe, D., Aublin, P., Kelbert, F., Reiher, T., Goltzsche, D., Eyers, D. M., Kapitza, R., Fetzer, C., and Pietzuch, P. R. Glamdring: Automatic application partitioning for intel SGX. In *USENIX Annual Technical Conference (ATC)* (2017), pp. 285–298.

[13] Liu, S., Tan, G., and Jaeger, T. Ptrsplit: Supporting general pointers in automatic program partitioning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2017), CCS '17, Association for Computing Machinery, p. 2359–2371.

[14] Liu, Y., Zhou, T., Chen, K., Chen, H., and Xia, Y. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *22nd ACM Conference on Computer and Communications Security (CCS)* (2015), pp. 1607–1619.

[15] Mambretti, A., Onarlioglu, K., Mulliner, C., Robertson, W., Kirda, E., Maggi, F., and Zanero, S. Trellis: Privilege separation for multi-user applications made easy. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)* (2016), pp. 437–456.

[16] Rubinov, K., Rosculete, L., Mitra, T., and Roychoudhury, A. Automated partitioning of Android applications for trusted execution environments. In *International Conference on Software engineering (ICSE)* (2016), pp. 923–934.

[17] Tilevich, E., and Smaragdakis, Y. J-orchestra: Automatic java application partitioning. In *European conference on object-oriented programming* (2002), Springer, pp. 178–204.

[18] Yongzheng Wu, Jun Sun, Y. L., and Dong, J. S. Automatically partition software into least privilege components using dynamic data dependency analysis. In *International Conference on Automated Software Engineering (ASE)* (2013), pp. 323–333.

[19] Zdancewic, S., Zheng, L., Nystrom, N., and Myers, A. Secure program partitioning. *ACM Transactions on Compututer Systems (TOCS) 20*, 3 (2002), 283–328.

[20] Zheng, L., Chong, S., Myers, A., and Zdancewic, S. Using replication and partitioning to build secure distributed systems. In *IEEE Symposium on Security and Privacy (S&P)* (2003), pp. 236–250.