Automatically Partition Software into Least Privilege Components using Dynamic Data Dependency Analysis

Yongzheng Wu and Jun Sun
Singapore University of Technology and Design
{yongzheng_wu,sunjun}@sutd.edu.sg

Yang Liu
Nanyang Technological University
yangliu@ntu.edu.sg

Jin Song Dong
School of Computing
National University of Singapore
dongjs@nus.edu.sg

***Abstract*—The principle of least privilege requires that software components should be granted only necessary privileges, so that compromising one component does not lead to compromising others. However, writing privilege separated software is difficult and as a result, a large number of software is monolithic, i.e., it runs as a whole without separation. Manually rewriting monolithic software into privilege separated software requires significant effort and can be error prone. We propose ProgramCutter, a novel approach to automatically partitioning monolithic software using dynamic data dependency analysis. ProgramCutter works by constructing a data dependency graph whose nodes are functions and edges are data dependencies between functions. The graph is then partitioned into subgraphs where each subgraph represents a least privilege component. The privilege separated software runs each component in a separated process with confined system privileges. We evaluate it by applying it on four open source software. We can reduce the privileged part of the program from 100% to below 22%, while having a reasonable execution time overhead. Since ProgramCutter does not require any expert knowledge of the software, it not only can be used by its developers for software refactoring, but also by end users or system administrators. Our contributions are threefold: (i) we define a quantitative measure of the security and performance of privilege separation; (ii) we propose a graph-based approach to compute the optimal separation based on dynamic information flow analysis; and (iii) the separation process is automatic and does not require expert knowledge of the software.**

I. Introduction

The principle of least privilege [29] requires that software components are only granted privileges that they need. If one component is compromised, the impact is minimized. Unfortunately, most software adopts the monolithic design, where the privileges of components are not separated. Any component is able to invoke procedures and modify states of any other components. Every component can access the same set of system resources such as files, network connections and hardware devices. If a bug in a component is exploited or a component is malicious, the whole software is compromised.

One of the reasons why the principle of least privilege is not widely applied on software components and monolithic design still dominates is because it is hard to find a good separation given the complexity of software. Firstly, it is hard [31] to separate software running in one process into multiple components running in multiple processes without shared memory. Secondly, the communication between components running in multiple processes is more costly comparing to single process version. Lastly, the privilege of different components must be sufficiently distinct so that the separation is meaningful.

Consider a scenario where the software provides some service to users connected from the Internet. The software uses PAM (Pluggable Authentication Modules) to authenticate each connected user by passing the user name and password to PAM. PAM then accesses the password database backend such as /etc/shadow or private keys and then returns back the authentication result. The software runs in a single process, thus compromising the software results in leaking /etc/shadow or private keys to the Internet. Even worse, since the process runs as the super user, the whole system can be compromised, resulting all files to be stolen. We can separate the software into three components each running in a separate process: an unprivileged component, a network component, and a PAM component. The unprivileged component performs most of the computation of the service, which requires neither access to the backend nor network. It presumably contains most of the software's logic and code base, thus is most likely to be buggy or vulnerable. The network component manages network connections and only accesses the network, but not the backend. The PAM component only accesses the backend. The unprivileged component can interact with the other components by calling their API functions. If the unprivileged component is compromised, it can neither read the backend nor send information through the Internet.[1]

Manually finding the cut among the three components requires understanding the software, which can be quite complex. A good cut should satisfy the following requirements. First, the code base of the high privilege components should be small so that they are less likely to be vulnerable. Second, the privilege of the high privilege components should be sufficiently distinct. In the previous case, a high privilege component can access either file or network, but not both. Third, the component interactions should be minimized so that the system is efficient.

[1]The compromised unprivileged component can still interact with other components through their API. For example, it can try all possible passwords in a brute force manner by calling the PAM API, but this can be easily limited by rate control.

978-1-4799-0215-6/13/$31.00 © 2013 IEEE

ASE 2013, Palo Alto, USA

We propose ProgramCutter, a tool to help software developers automatically refactor monolithic software into least privilege components which run in different processes. ProgramCutter finds the optimal cut according to the three requirements above. In addition, given the source code of monolithic software, ProgramCutter automatically transforms it into the component separated form. ProgramCutter does not require expert knowledge of the software, although additional knowledge can give better results.

ProgramCutter works by analyzing execution traces of the software to get the communication and privilege of each function. We consider two forms of communications between functions: function invocation and data dependency. Function invocation is self-explanatory. A function f_1 is data-dependent on f_2 if f_1 reads from the data that is previously written by f_2. The communication among functions forms a graph, where the functions are nodes and the communications are edges. Each function invocation and data dependency has a communication cost which is reflected on the edge weight. Each function is associated with certain system privileges if it makes certain system calls. Each node is also associated with a weight reflecting the size of the code of the function body. ProgramCutter finds the optimal separation of components by searching for a partition of the graph such that: nodes with different privileges are in different partitions; inter-partition edge weight is minimized; and the total node weight of high privilege component is minimized. Finally, ProgramCutter refactors the software into a privilege separated form by changing inter-component calls to remote procedure calls.

We have implemented a prototype and evaluated it by separating four open source software. The evaluation shows that we can bring about 90% of the code from running in privileged mode to unprivileged mode. This means that most of the vulnerabilities are contained in the unprivileged component, thus they are mitigated by ProgramCutter. The cost paid in performance overhead is below 20% except in a rare extreme case.

II. Related Work

Manual Separation: Privilege separation has been manually applied on many software to prevent attackers from gaining full privilege. Service software (daemon) such as postfix [1], sendmail [2] and vsftpd [3] typically first executes as the super user in order to acquire system resources such as binding network ports and opening log files for writing. After that, it gives up all privileges so that in case it is compromised, it cannot acquire additional system resources. Service software that constantly requires privilege cannot adopt this method. For example, OpenSSH needs to call `setuid(2)`, which requires root privilege, after each successful user authentication in order to launch a shell on behave of the user. Provos et al. manually separated OpenSSH [28] into a high privilege process which only does authentication and a low privilege process which does everything else. Doing privilege separation manually can take a substantial amount of time and can be error prone.

Automated Separation: Kilpatrick proposed Privman [17] which provides reusable library to help software developers to write privilege separated software. Brumley et al. proposed Privtrans [9] which provides some degree of automation in privilege separation. Software developers only need to specify privileged functions and variables by adding annotations in the source code, and Privtrans generates source code of privilege separated software. Although both approaches provide some degree of automation, expert knowledge on the software is still required. ProgramCutter does not require any expert knowledge. Other than that, the main difference between Privtrans and ProgramCutter is that Privtrans uses static analysis whereas ProgramCutter uses dynamic analysis. The problem with static analysis is that firstly, dynamic behavior such as type cast, function pointer and pointer arithmetic cannot be accurately analyzed by static analysis. Secondly, runtime information such as buffer size is not available for static analysis. As a result, Privtrans prefers the size of high privileged component to be minimal and ignores the communication cost. ProgramCutter generalizes this by considering communication cost and allowing the user to balance the trade-off between the two. Lastly, software developers are still required to understand the software in order to pinpoint the privileged functions and variables. On the other hand, dynamic analysis has the downside of incomplete coverage. The accuracy of dynamic analysis depends on the completeness of the execution traces. We will further discuss this issue in Section III-A.

Apart from separating C programs, Swift [12] works on a Java-like language, named Jif, in which programmers explicitly classify variables in order to specify a information flow policy such as confidential information should stay in the web server and not be send to the web browser. Swift is then able to partition the program into two programs one running in the web server and the other in the web browser, while ensuring the information flow policy. Both Swift and ProgramCutter adopt a graph cut based approach, where the edge weight represents the amount of information flow. However, Swift adopts a static approach while ProgramCutter adopts a dynamic one. To determine the amount of information flow, we need to know the number of times a piece of code is executed, which is unavailable in static analysis. Swift uses certain assumptions such as equal probability of both branches in all `if` statements. On the other hand, ProgramCutter relies on execution traces. The lack of dynamic features such as pointer arithmetic, function pointers and unrestricted type casting in Jif (or Java) make static information flow analysis simpler.

Separation Mechanisms: Implementation of separation should ensure software components only (i) access their dedicated memory, (ii) execute their dedicated code and (iii) only call the granted system calls. There are two commonly adopted approaches: verification-based and OS-based. Verification-based separation [30], [24], [25] ensures that there is no bad instruction, which accesses memory beyond the dedicated memory range. This is done by scanning all memory loading and storing instructions to make sure that the target address falls inside the dedicated memory range.

For indirect memory access, where the target address is only known at the runtime, an assertion is inserted before the instruction. OS-based separation [26], [9], [17], [28] runs different components in different processes and uses the hardware paging mechanism to ensure the memory safety, and therefore there is no performance penalty on memory access as in verification-based separation. However, the inter-component calls are implemented as IPCs, thus it has performance penalty proportional to the number of inter-component calls and size of parameter passed. It is feasible to adopt either approach. However, as our component separation tries to minimize inter-component communication, the OS-based separation suits us better.

Software Module Clustering: Graph-based approach has been used to partition software module for the purpose of program comprehension. Bunch [23] models software into a graph where the nodes represent modules and the edges represent their dependencies such as function invocations and variable access. It then searches for the optimal cut that has least inter-partition edge weight. While both Bunch and ProgramCutter adopt a min-cut algorithm on searching for the optimal cut, ProgramCutter incorporates security properties of software components and the graph cut has to satisfy the constrain of the security policy. In addition, the edge weight in ProgramCutter represents information flow (dynamic), while it represents module dependency (static) in Bunch.

III. System Design

Separating software using ProgramCutter consists of three stages. As illustrated in Fig. 1, a monolithic program is first compiled with debug symbols. The program is then executed in the *trace collector* which collects execution traces. The traces are then analyzed by the *graph partitioner* in order to compute the optimal separation of components. The separated components are used by the *source translator* to rewrite the monolithic program into a privilege separated program where different components run in different processes.

When using ProgramCutter, we do not assume that the user has any knowledge of the program to be separated, hence we claim that ProgramCutter separates programs *automatically*. However, we assume that the user has some knowledge on system privileges, which are independent of the program itself.

To describe how ProgramCutter works, we separate a toy program shown in Fig. 2. The program signs the message "I am [user]" for an authenticated user. The user launches the program by providing his user name and password. The program then employs a password-based authentication method, which reads the system password database /etc/shadow. After a successful authentication, it signs the message "I am [user]", where "[user]" is the authenticated user, using a private key. For simplicity, we omit the error handling and cleaning code. We also omit the code of function `matches()` and `dosign()`. `matches(l,u,p)` matches password `p` together with user name `u` with an entry `l` in the password database. `dosign(p,m)` signs message `m` using private key `k` and returns a signature. The program reads two confidential files, `/etc/shadow` and `/private_key`. We purposely introduced a buffer overflow vulnerability in Line 27. This is a typical stack-based buffer overflow vulnerability [27] which can lead to execution of arbitrary code. We will later demonstrate how ProgramCutter automatically separates the program into three components with different privileges, so that in case the vulnerability is exploited, the confidential files cannot be stolen by the attacker.

A. Trace Collector

The trace collector logs execution traces that record memory access and privileged system calls done by each function. More specifically, each trace consists of a sequence of events, where each event is either a memory access or a privileged system call.

- **Memory Access:** We record all memory read and write operations together with the virtual address range. This includes all memory accesses to the program stack and heap. Each record is in the form of (function, operation, address and size) tuple. Only user space memory accesses are recorded. The recorded functions are the functions that perform the memory operations. The operations can be either reading or writing. The address and size capture the target memory range. We only record memory addresses but not variable names because using debug symbol of the program, we can translate memory addresses to variable names.
  In addition to instructions that read or write user space memory, we also record memory access to user space memory due to system calls. For example, the system call `write(fd, ptr, size)` writes N bytes of user space memory from `ptr` to file descriptor `fd`, where N is the return value. In this case, we record that memory $[ptr, ptr + N)$ is read.
  Table I shows some of the memory access records generated by our authentication program. The order of the records is significant. Note that the last column, line#, which is the program line number responsible for the memory access, does not appear in the record and is added manually for the purpose of the illustration.
- **Privileged System Call:** In this paper, we assume that all privileged operations are made by system calls, such as opening a file, sending data to a remote host. To have fine grained privilege, we record system call parameters such as file path and network address.
  After traces are collected, the user lists all privileged system calls and labels some of them. In the end, the program will be partitioned into lease privileged components according to the labels. Labeling privileged system calls does not require understanding the structure of the software, because only the knowledge on system privileges is required.
  In our toy program, reading from /etc/shadow or /private_key is considered privileged. The user may use one label to label reading from the two files. He may use two labels to label them separately to give more fine grained

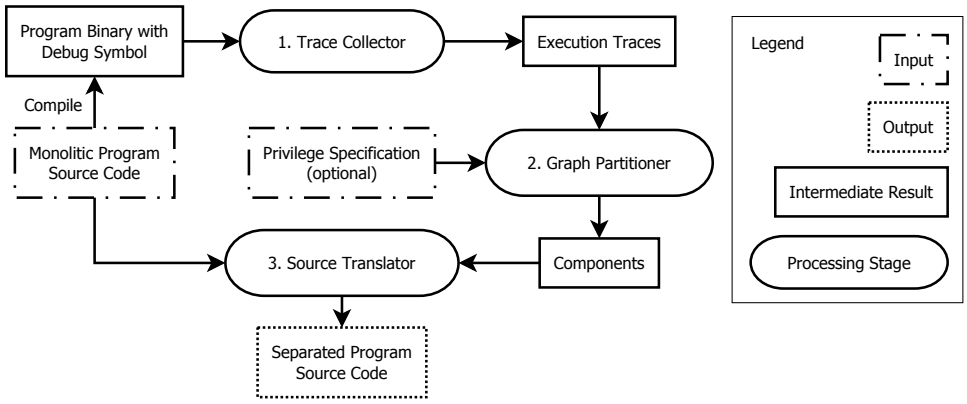

Fig. 1. Overview of ProgramCutter stages

TABLE I
MEMORY ACCESS RECORDS

function	operation	address	size	line#
main	write	`username`	pointer size	23
inpasswd	write	`line`	BUFSIZE (by syscall)	5
inpasswd	write	return value	word size	7
main	read	return value	word size	25

separation. In the latter case, the program will be separated into three components: an unprivileged component, a component for reading /etc/shadow, and a component for reading /private_key.

For another example, in OpenSSH server, the user may use three labels. Opening /etc/ssh/ssh_host_rsa_key is labeled *privatekey*; network related system calls are labeled *network* and other file writing except in /tmp are labeled *filewrite*. In the end, it will be partitioned into four components.

We can collect traces from multiple executions in order to get sufficient coverage of the program. Multiple executions can be collected by running the test cases given by the program developer or from other automated test case generation methods [13], [18], [10].

B. Graph Partitioner

The task of graph partitioner is to partition the set of functions into a number of components, such that different privileges are separated into different components. In addition, there is an unprivileged component, which cannot call any privileged system call. As mentioned earlier, the code size of privileged components[2] should be minimized, as well as the communication among components.

We propose a novel graph-based formulation of this problem. A function f_1 depends on f_2 if f_1 reads memory that is last written by f_2. If we partition f_1 and f_2 into two different components, we need to transfer the memory from one component to the other, which incurs communication cost. In our graph, each node denotes a function. The edge weight between two nodes is the sum of the number of bytes of the memory dependency between the two functions. This captures the communication cost if the two functions are partitioned into different components. Each node is associated with a weight denoting the size of the source code implementing the function. This captures the size of the code base of a component, which is correlated to the probability of being buggy or vulnerable. If a function performs a privileged system call, the corresponding node is labeled with the privilege of the system call. Functions that do not perform system calls are not labeled, except that the program entry point, `main()`, is assigned with a special unprivileged label[3]. The task of the graph partitioner is to partition the graph into N partitions, where N is the number of different labels, such that (i) all labeled nodes in each partition share the same label; (ii) the total weight of the inter-partition edge is minimized; and (iii) the total weight of privileged partitions is minimized. Since we have two values to minimize, we aggregate the two into one value by computing a weighted sum. The weight is used to balance between security and performance.

Fig. 3 shows the data dependency graph of our toy program. The dependency between `main` and `inpassword` is the size of user name and password, which is typically less than 40 bytes. However, the dependency between `inpassword` and `fgets` is the size of /etc/shadow file, which is much larger. The graph partitioner may partition the `inpassword` function in the privileged component in order to minimize communication cost. In this way, the graph partitioner auto-

[2]For simplicity, we consider all privileges to be equally significant. An alternative would be having a significance factor for each privilege and minimizing the weighted sum of the code size.

[3]We assume `main()` does not directly perform any privileged system call. If it does, as the case in Section VI-B, we manually make a wrapper for the system call.

```
1  static int inpasswd (char *username, char *password)
2  {
3    char line[BUFSIZE];
4    FILE *fp = fopen("/etc/shadow", "r");
5    while (fgets(line, BUFSIZE, fp) != NULL) {
6      if (matches(line, username, password))
7        return 1;
8    }
9    return 0;
10 }
11
12 static char *signmsg (char *message)
13 {
14   char privkey[KEYSIZE];
15   FILE *fp = fopen("/private_key", "r");
16   fread(privkey, KEYSIZE, 1, fp);
17   char *signature = dosign(privkey, message);
18   return signature;
19 }
20
21 int main (int argc, char **argv)
22 {
23   char *username = argv[1];
24   char *password = argv[2];
25   if (inpasswd(username, password)) {
26     char buffer [100];
27     sprintf(buffer, "I am %s", username);
28     char *signature = signmsg(buffer);
29     printf("%s\n", signature);
30   } else
31     printf("Bad login\n");
32 }
```

Fig. 2. A toy program that signs a message for an authenticated user

matically finds out the function in charge of authentication. Similarly, the data dependency between `main` and `signmsg` is the size of the message and the signature, which is smaller than the cryptographic key. Thus the graph partitioner may partition the `signmsg` function in the privileged component.

This problem is a variant of multi-terminal cut problem, whose complexity is in general NP-hard and was first extensively studied by Dahlhaus et al. [14]. A number of solutions and approximations were proposed [11], [16]. Our main focus is not to solve the multi-terminal cut problem, instead, we can apply any existing solution. In particular, we choose an integer programming based solution which will be described in Section IV. Our choice of solution is easy to implement and its performance is sufficient for our purpose, which will be shown in Section VI.

In the end, the graph partitioner outputs the labels to all functions. In our toy example, if we define two labels: *private key* for reading `/private_key` and *user password* for reading /etc/shadow, the graph partitioner will assign `main`, `signmsg` and `inpassword` to the unprivileged label, *private key* and *user password* respectively.

C. Source Translator

The output of the graph partitioner can be manually examined by the software developer to help studying code structure or to find out the code that is responsible for privileged operations. In this way, ProgramCutter is used as a program comprehension tool. We take a step further to automatically translate the original program into privilege separated program.

The privilege separated program works as follows. The program consists of L processes, where L is the number of labels including the unprivileged label. As `main` is always assigned the unprivileged label, the program starts in the unprivileged process, which cannot directly perform any privileged system call. Rather, it needs to perform privileged operations through the privileged processes by doing remote procedure calls (RPCs). The privilege separated program firstly launches L processes, where the unprivileged process start the execution

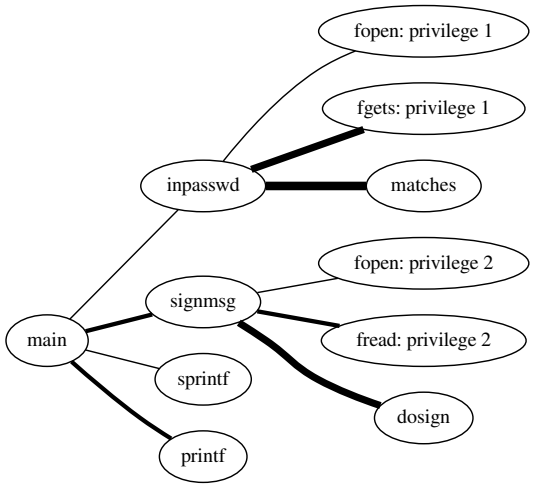

Fig. 3. The data dependency graph of our toy program. The thickness of an edge represents the amount of the data dependency between two functions. We mark two sets of privileges: 1. reading `/etc/shadow` and 2. reading private key file.

from `main()` and other processes wait to be called by RPCs. The inter-component function calls are translated to RPCs. Each component is allowed to call a limited set of RPCs in other components. This set is determined by outgoing edges of each partitioned component in the graph.

In the separated toy program, three processes representing three components are created. The unprivileged process executes `main()`, while the two privileged processes wait for RPC. The invocation of `inpassword()` on Line 25 is replaced by a RPC invocation. After the unprivileged process invokes the RPC, the execution is transferred to the privileged process, which then executes `inpassword()` and returns the result back to the unprivileged process through the RPC return. Similarly, The invocation of `signmsg()` on Line 28 is replaced by a RPC invocation as well.

When the buffer overflow vulnerability on Line 27 is exploited, the attack can execute arbitrary code in the unprivileged process. This process cannot read `/etc/shadow` or `/private_key`. It can invoke the two RPCs. However, by invoking `inpassword()`, it cannot read the password database either. By invoking `signmsg()`, it can sign arbitrary messages, but cannot read the private key. In this case, we mitigate the damage from losing private key to signing arbitrary messages.

Difficulty on Parameter Marshalling: RPC requires parameters to be passed-by-value, because passing-by-reference does not work when the caller and callee are in two different address spaces. Our source translator translates passing-by-reference function call to passed-by-value using parameter marshalling. Unfortunately, parameter marshalling requires the type of each parameter to be serializable, which cannot be done automatically, hence the serialization function has to be manually implemented for each data type.

We manually implemented the serialization functions for C library's commonly used data types such as `FILE` and `time_t`. In addition, our source translator deals with the following types.

- *Buffer with Statically Known Size:* This includes character arrays such as the `buffer`, `privkey` and `line` variables in our toy program.
- *Structure with no Pointers:* A structure's size is statically known thus it can be handled similar to a buffer. However, if a structure contains pointers, which may be dereferenced by the other process, we need to marshal the pointed data. This can be problematic if the other process modifies the pointer.
- *Null-terminated String:* Although the length of a string is not statically known, we can compute it at runtime as long as we know it is a null-terminated string. We use some heuristic to determine whether a character array is a string. (i) Strings are passed as a parameter to a known library function. For example, if variables `x` and `y` are passed to `fopen(x, y)`, we know that both `x` and `y` are null-terminated strings. The variables `username` and `password` are recognized as string in this manner. (ii) Similarly, Strings are returned from a known library function, such as `strdup`.

For other custom types, the serialization functions have to be manually implemented in order to use our source translator. Implementing the serialization functions requires much little work than manually separating the program.

IV. Implementation

We choose to implement ProgramCutter in GNU/Linux since there are more open source software to test with. It can be easily ported to other operating systems since we do not rely on any special feature of Linux.

Trace Collector: The trace collector is implemented using Pin [22]. Pin works by dynamically instrumenting instructions of a program binary in run time. We need to collect information on two types of events, memory reading/writing and system call. The former is done by instrumenting each memory reading/writing instruction and recording the instruction pointer, memory range during execution. The latter is done by hooking all system call entry and exit points[4] and recording the instruction pointers that invoke the system call, parameters and return values. ProgramCutter currently recognizes the following privileged system calls: file opening/reading/writing, socket creation/binding/connecting/sending/receiving, program execution, changing user ID and sending signal. Additional privileged system calls can be easily added. We only record the instruction pointer but not the name of the function because we can look up the name of the function using instruction pointer from the debug symbol.

We only record functions of the program but not all functions including system libraries because ProgramCutter

[4] `PIN_AddSyscallEntryFunction` and `PIN_AddSyscallExitFunction` are PIN APIs to hook system calls.

partitions the program, not system libraries. As a result, when the instruction pointer does not fall within the program's code range, we need to backtrack and find out the latest caller in the program and record the instruction pointer of that call invocation. For example, in our toy example, Line 5, `fgets`[5], which is a C library function, writes to the buffer pointed by `line`. Instead of recording the instruction pointer of `fgets`, we record its caller in the program, `inpasswd`.

Graph Partitioner: The graph partitioner consists of two stages, constructing the graph and partitioning the graph. The graph can be constructed by scanning the trace in a single pass. Throughout the algorithm, a data structure is maintained to keep the last writer of each byte in the memory. Initially, the last writer of the whole memory is *none*, meaning uninitialized. When scanning a memory writing record, the last writer of all the bytes in the record's memory range is changed to the function of the record. When scanning a memory reading record, for each byte in the record's memory range, we add an edge, or increment the edge's weight if already added, between the reader and the last writer. In the end, we obtain a graph where each node represents a function and each edge represents the data dependency. Instead of keeping the last writer of individual bytes, we use a redblack tree to store segments of memory. The running time complexity is $O(n \log k)$, where n is the number of records in the trace and k is the number of memory segments, which is much smaller than n in practice.

After we have obtained the graph, we reduce the multi-terminal cut problem to an integer programming problem and use *lp_solve* [8] to solve it. We choose this approach because it is easy to implement and works reasonably well.

We reduce it into an integer programming problem as follows. For each function, we declare an integer variable $l_{1...F} \in [1, L]$ (where L is the number of labels and F is the number of functions) to denote the label to be assigned to the function. We use value 1 to denote the unprivileged label and value above 1 for privileged labels. For pre-labeled function i, l_i is fixed. For an edge i with weight w between function j and k, we declare a variable $e_i = w$ if $l_j \neq l_k$ and $e_i = 0$ if $l_j = l_k$ to represent the cost of cutting it. For each function i with weight w, we declare a variable $f_i = w$ if $l_i \neq 1$ and $f_i = 0$ if $l_i = 1$ to represent the cost of putting the function in a privileged partition. The value to be minimized is weighted sum of (i) the sum of all edge costs and (ii) sum of all functions: $\alpha \sum_{i=1}^{E} e_i + (1-\alpha) \sum_{i=1}^{F} f_i$, where α is the weight that can be adjusted. In total, $F\lceil log_2(L) \rceil + E$ variables and $2E\lceil log_2(L) \rceil$ constrains are declared, where F is the number of unlabeled functions and E is the number of edges.

Source Translator: To make different components execute in different processes, there are two major changes to the program. First, before invoking `main`, (i) the additional processes have to be created; (ii) their system privileges have to be confined according to the privilege specification; and (iii) the communication sockets have to be initialized. Confining system privilege can be enforced by capability systems [20] or system call monitoring [19], [15], which are orthogonal to ProgramCutter. Second, inter-component function calls have to be changed to RPCs.

We use a substituted `main` to perform the initialization. The original `main` is renamed to `orig_main`[6]. If the process is the main process, the substituted `main` calls `orig_main` after the initialization. Otherwise, after initialization, the substituted `main` enters an event loop, in which the process waits and dispatches incoming RPCs.

We say a function to be a *component entry* function if it can be called from another component. We substitute each component entry function with an RPC wrapper, which handles the parameter marshalling and communication. We also check if the calling component is allowed to call the entry function at runtime. The original function is renamed to `orig_`*`func`*. The calling of a component entry function (e.g. Line 25 and 28 in Fig. 2) is unchanged.

In our toy example, the function names in the declarations on Line 1, 12 and 21 are rewriten with prefix `orig_`. The substituted `main` performs initialization as described earlier and either invokes `orig_main` or enters RPC dispatching loop. Since `inpasswd` and `signmsg` are component entry functions, we add one RPC wrapper for each function.

V. Security Analysis

In our threat model, we assume the privileged component to be vulnerability-free, while the unprivileged component to be vulnerable and is compromised to execute arbitrary code. This assumption is based on the fact that (i) the number of software bugs is positively correlated to the size of code, and (ii) ProgramCutter makes the privileged component to be as small as possible. Previous study has built different models to correlate the number software bugs and the software size. Akiyama [7] found that the lines of code (L) and the number of bugs (B) are linearly correlated, while Lipow [21] found B to follow a quadratic function of $log(L)$. We also assume that the attacker is aware of ProgramCutter and knows how the program is partitioned. Our goal is to prevent the attacker from performing privileged operations that are specified to the graph partitioner. We now discuss the potential attacks and how ProgramCutter prevents (or fails to prevent) them.

- **System Call Invocation:** The compromised unprivileged component may try to directly invoke privileged system calls. Since the unprivileged component runs in an unprivileged process, the invocation is denied by the system call filtering mechanism.
- **Modifying Privileged Code/Data:** The compromised component may try to change the code of the privileged component and let it perform privileged operations on behave of the attacker. The compromised component may modify data to achieve similar purposes. For example,

[5]More precisely, `_IO_getline_internal`, which is called by `fgets`, performs the memory write.

[6]For simplicity, we assume that there is no function named `orig_main` in the original program.

it can change function pointers or other control flow related data structures. This is prevented because different components run in different processes and they do not have directly access each other's memory.

- **Exploiting Privileged Component Entry Function:** An unprivileged component is permitted to call entry functions of another component. If the entry functions, in turn, invoke privileged system calls, the attacker can call it to perform attacks. For example, if the task of an entry function is to append a string to a log file, and both the string and log file are specified as function arguments, the attacker is able to append arbitrary data to an arbitrary file. This can be exploited to attack the file system. One way to prevent this type of attacks is to do input sanitization in the entry functions. ProgramCutter does not automatically perform input sanitization, thus manual work is needed.
- **Collusion Attack:** Two or more compromised components can collude to perform attacks that cannot be achieved by a single component. For example, if both the network component and sensitive data component are compromised by the same attacker, sensitive data can be sent to the network. Two conditions must be met in order for the attack to succeed. First, there must be a usable entry function. For example, there must be an entry function in the network component that can be used to send data to the attacker. Similar to previous attack, input sanitization can be used to eliminate this condition. Second, the entry function must be directly callable from the other component. ProgramCutter fails to prevent collusion attack if and only if both conditions are satisfied.
- **Denial of Service:** The compromised component can pass incorrect data or refuse to do any useful work. This can cause the program to behave incorrectly or crash. ProgramCutter does not ensure correctness thus does not prevent this type of attacks. Another form of DoS attack is that the compromised component repeatedly calls the trusted component so that the trusted component is not able to serve other legitimate components. This attack can be effectively mitigated by thresholding the calling frequency based on the calling component if the expected frequency is known.

VI. Evaluation

In this section, we use ProgramCutter to partition real-word programs. We want to evaluate ProgramCutter based on the following criteria.

- The trace collector should be able to monitor and collect all function calls, system calls and memory operations of the program. The execution time overhead of the program being monitored and the size of the trace should be practical.
- The graph partitioner should be able to construct the graph and solve its optimal partition within a reasonable time.
- The partitioned program should behave identical to the original program in normal execution. In addition to run the program in the same configuration (command line options and input) as the one generating the trace, we should be able to run it in different configurations. The performance overhead should be reasonable.
- The size of the privileged components in the partitioned program should be small. We also want to know if real-world security vulnerabilities can be mitigated.
- If a bug in an unprivileged component is exploited to execute malicious code, system privileges should not be used by the component. In particular, we should be able to prevent the attacker from gaining a root shell.[7]

We evaluate ProgramCutter using the following software: OpenSSH server, wget, ping and thttpd, because they (i) use system privileges (especially OpenSSH server and ping run as root) which make them to be attacker's valuable targets; (ii) read from external input which can an attacking vector; and (iii) are widely used and previously known to have security vulnerabilities.

For each of the four programs, we first describe how we execute it to collect the execution trace. We then show statistics of the program and trace. We label the privileged system calls and use the graph partitioner to partition the program into components. Finally, we use the source translator to obtain the privilege separated programs. We test the privilege separated programs from three aspects: (i) We execute them several times with different configurations or input to see if the separated program behave like the original program. (ii) We compare the execution time of the two programs. (iii) We execute the privilege separated programs with root privilege and try to exploit their vulnerabilities to see if we can obtain a root shell or read confidential information such as user passwords and private keys.

We quantitatively evaluate the quality of the partition in terms of security and performance. We measure the security by the size of code executed in unprivileged process. The original program executes all code in the privileged process. The privilege separated program executes only a fraction of the code in the privileged process. The smaller the fraction is, the more secure the program is. We look at the performance overhead which is the additional execution time divided by the original execution time. In all our test cases, we set the weight α in the optimization algorithm to be 0.5, which we obtain through experiments. A summary of the statistics and benchmark results are listed in Table II. All the benchmarks are performed on an Intel Core i5-2520M machine with 4G physical memory running 32-bit Fedora 17 (kernel 3.5.4).

A. OpenSSH Server

OpenSSH server is the most widely used SSH server in Unix-based systems. We collect an execution trace of the server by connecting to the server, performing a password

[7] A root shell is a shell with root privilege. Gaining a root shell is commonly the first thing to do after a security vulnerability is exploited.

TABLE II
SUMMARY OF EVALUATION RESULTS. ABBREVIATIONS: "FUNC.": TOTAL NUMBER OF FUNCTIONS APPEARED IN THE TRACE; "LOC": TOTAL LINES OF CODE IN THE CORRESPONDING FUNCTIONS; "F-INV": TOTAL NUMBER OF FUNCTION INVOCATIONS IN THE TRACE; "COMP.": NUMBER OF COMPONENTS TO PARTITION INTO; "P-LOC": TOTAL LOC IN THE PRIVILEGED PARTITIONS; "t_{part}": THE RUNNING TIME OF THE GRAPH PARTITIONER; "t_{mono}": THE EXECUTION TIME OF THE ORIGINAL MONOLITHIC PROGRAM; AND "t_{sep}": THE EXECUTION TIME OF THE SEPARATED PROGRAM.

software	func.	LOC	f-ivk.	comp.	p-LOC	t_{part}	t_{mono}	t_{sep}
openssh	219	10244	770763	3	563 (5.5%)	7.771s	153ms	181ms (+18%)
ping	11	2149	7369	2	304 (12%)	1.133s	10000.196s	10000.281s ($+10^{-4}$%)
wget	197	13156	22305	2	791 (6%)	2.150s	815ms	830ms (+1.8%)
thttpd	38	2717	7492	2	617 (22%)	0.859s	815ms	815ms (0%)

based authentication, and immediately logging out from the server. The source code of OpenSSH server (version 3.1p1) consists of 27419 lines of code (LOC) and 504 functions. In our trace, 219 different functions are invoked 770763 times in total. We define two privileged labels: *shadow file* for reading the user password file and *private key* for reading the private key files. The graph partitioner partitions the 504 functions into three components. The *shadow file* partition has 12 functions with 401 LOC; the *private key* partition has 3 functions with 162 LOC; and the unprivileged partition has the rest.

The execution time of the original monolithic server is 0.153 seconds and the time of separated server is 0.181s (18% more). Note that the execution time here is an average of 10 executions. To minimize the network delay, we execute both the SSH server and client on the same host and use the local loop back interface as the network medium, Thus, with actual networks, the performance overhead should be much smaller than 18%. Since 95.5% of the code is in the unprivileged component, most of the vulnerabilities, such as CVE-2003-0682 [5] and CVE-2003-0695 [6] are due to bugs in the unprivileged component, thus they can be mitigated by ProgramCutter. For example, CVE-2003-0695 exploits a bug in `buffer_init` which is in the unprivileged component, thus it cannot access the shadow file or private key.

We test the following different execution scenarios and found the privilege separated SSH server and the original server behave identical. (i) The client authenticates with different accounts. (ii) Try two different client versions (5.9p1 and 3.1p1). (iii) Try different server key sizes (1024, 2048 and 4096 bits). Note that we use the same privilege separated SSH server to test all scenarios. This shows that our trace has sufficient coverage in order to support different execution scenarios.

B. ping

Ping is a network diagnostic program which sends ICMP echo packets to a remote host and receives reply from it. Since sending ICMP packets requires super user privilege, the ping tool is a setuid program, which means that the program always runs in super user privilege regardless of the user who runs it. This makes `ping` to be a highly wanted program by attackers to find vulnerabilities. For example, the vulnerability CVE-2000-1214 allows normal user to use super user privilege to execute arbitrary code.

Ping is a simple program with only 11 functions in 2149 LOC. We execute ping (version s20101006) to send and receive 3 packets in our trace collector. 7369 function invocations are recorded in the trace. We label the system call of sending and receiving ICMP packets as privileges. We found that the `main` function directly invokes the privileged system calls `socket`, `setsockopt`, `ioctl` and `bind`. This makes ProgramCutter unable to partition the single function. We thus add four wrapper functions for them. The graph partitioner partitions the program into two components. The privileged component contains 3 functions with 304 LOC from the original program and 4 wrapper functions.

The execution time overhead is too small to be measured because ping waits for one second between sending each packet by default. The variance of the execution time caused by random factors is larger than the overhead we want to measure. In order to minimize the base execution time, we set the wait interval between sending each packet to 0 (flood mode) and use the local loopback interface as destination. The time of sending 10000 packets using the original monolithic program is 0.196 seconds; while the time using the separated program is 0.281 seconds (43% more). Using this timing, we can estimate the execution time overhead with the default one packet per second. The execution time of the monolithic program to send 10000 packets is $t_{mono} \approx 10000 + 0.196 = 10000.196$, and the time of the separated program program is $t_{sep} \approx 10000 + 0.281 = 10000.281$. Thus, the overhead is $(t_{sep} - t_{mono})/t_{mono} = 8.5 \times 10^{-6}$.

We test different scenarios such as running in verbose and quiet modes, using different packet sizes, and sending to different addresses. The separated program mitigates CVE-2000-1214[4] such that the attacker cannot gain super user privilege.

C. wget

Wget is a command-line program to download files from the Internet. We want to partition it into a privileged networking component and an unprivileged component. We collect our traces by using wget (version 1.13.4) to download a file from a web server. 197 different functions are invoked in 22305 invocations. The privileged system calls performed are `bind`, `connect`, `socket`, `read`, `write`[8], `recv` and `send`. The

[8] `read` and `write` can operate on both local files and sockets. Here, we are only interested in sockets.

privileged partition has 10 functions with 791 LOC and the unprivileged partition has 187 functions with 12365 LOC.

To measure the execution time, we download a 1MB file 100 times from a local web server. The execution time of the original monolithic wget is 0.815 seconds and the time of the separated wget is 0.830 seconds (1.8% more). We test different scenarios using different URLs, file sizes and enumerating different command line options such as verbose mode, continuing downloading previously partially downloaded files, and using a web proxy. The privilege separated wget behaves identical to the original one.

D. thttpd

`thttpd` is an open source web server. Similar to wget, we want to partition it into a privileged networking component and an unprivileged component. We collect our trace by letting thttpd (version 2.25b) serve one file download request. 38 different functions are invoked in 7492 invocations. The privileged partition has 8 functions with 617 LOC and the unprivileged partition has 30 functions with 2100 LOC.

To measure the execution time, we use it to serve a 1MB file downloading 100 times by a local client. The execution time of the original monolithic thttpd is 0.815 seconds with standard deviation as large as 0.733 seconds which is much larger than the overhead (less than 10ms). As a result, the overhead is not measurable with practical number of tests. We test the privilege separated `thttpd` with different configuration options such as different document root, logging option, cache policy. We also try to use different web browsers to access the server. The two copies behave identical.

VII. Limitations

We now discuss the limitations of ProgramCutter.

- Our dependency graph is based on execution traces, thus the correctness of the graph partitioner depends on the completeness of the execution traces. Although we cannot guarantee our dynamic approach to be complete, we can see from our evaluation that different execution scenarios are covered. In addition, works [13], [18], [10] in the literature have made progress in generating test cases to get good code coverage.
- As discussed in Section III-C, the source translator can only automatically handle parameter passing of generic data structures. Manual work is needed in implementing marshalling of application defined data structures. However, the amount of manual work is substantially smaller than rewriting the program in a different programming language as in Swift [12]. Also, the person who implements the marshalling is only required to understand the data structure to be marshalled, but not the full program to be partitioned.
- If an unprivileged component is compromised, it is able to invoke RPC to privileged components. The potential damage to the system depends on the functionality of the RPC. For example, if the entry function of a privileged component does not sanitize function parameters, it can be exploited to execute arbitrary code in the privileged component.
  There are two ways to prevent this. Firstly, the programmer can sanitize the function parameters of the entry functions, so that they cannot be abused by the caller. Secondly, a fine grained system privilege can be applied to restrict the privileged component, so that even if it is exploited, the damage is limited.
- We adopt a function-based separation, where a function is the base unit. Thus, we cannot partition a program which has only one function. A workaround is to make a wrapper function for the privileged system calls and perform proper parameter sanitation to prevent abuse. This is demonstrated in the `ping` case in Section VI.

VIII. Conclusion

In this paper, we present ProgramCutter, a novel approach to automatically partitioning software into least privilege components using dynamic data dependency analysis. The partition is optimal in terms of our quantitative measure of security and performance. The separation process is automatic and does not require any expert knowledge of the software. Our evaluation shows that we can take most of the code from running in privileged mode to unprivileged mode with reasonable performance overhead.

IX. Acknowledgment

This work is supported by project "IDD11100102A/ IDG31100105A" from Singapore University of Technology and Design, and NTU-NAP project "Formal Verification on Cloud".

References

[1] Postfix home page. http://www.postfix.org/.
[2] Sendmail home page. http://www.sendmail.org/.
[3] vsftpd home page. https://security.appspot.com/vsftpd.html.
[4] CVE-2000-1214: Buffer overflows in the (1) outpack or (2) buf variables of ping in iputils before 20001010. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2000-1214, 2000.
[5] CVE-2003-0682: "Memory bugs" in OpenSSH 3.7.1 and earlier. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2003-0682, 2003.
[6] CVE-2003-0695: Multiple "buffer management errors" in OpenSSH before 3.7.1 may allow attackers to cause a denial of service or execute arbitrary code. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2003-0695, 2003.
[7] Fumio Akiyama. An example of software system debugging. *Information Processing*, 71(1):353–379, 1971.
[8] M. Berkelaar, K. Eikland, and P. Notebaert. lpsolve: Open source (mixed-integer) linear programming system. *Eindhoven U. of Technology*, 2004.
[9] D. Brumley and D. Song. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security*, pages 57–72, 2004.
[10] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX OSDI*, pages 209–224, 2008.
[11] G. Calinescu, H. Karloff, and Y. Rabani. An improved approximation algorithm for multiway cut. In *STOC*, pages 48–52, 1998.
[12] Stephen Chong, Jed Liu, Andrew C Myers, Xin Qi, Krishnaprasad Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 31–44. ACM, 2007.
[13] L.A. Clarke. A system to generate test data and symbolically execute programs. *TSE*, (3):215–222, 1976.

[14] E. Dahlhaus, D.S. Johnson, C.H. Papadimitriou, P.D. Seymour, and M. Yannakakis. The complexity of multiterminal cuts. *SICOMP*, 23(4):864–894, 1994.

[15] M. Haardt and M. Coleman. ptrace(2) manual, 1999.

[16] D.R. Karger, P. Klein, C. Stein, M. Thorup, and N.E. Young. Rounding algorithms for a geometric embedding of minimum multiway cut. In *STOC*, pages 668–678, 1999.

[17] D. Kilpatrick. Privman: A library for partitioning applications. In *FREENIX*, volume 8, 2003.

[18] B. Korel. Automated software test data generation. *TSE*, 16(8):870–879, 1990.

[19] A. Kurchuk and A. Keromytis. Recursive sandboxes: Extending systrace to empower applications. *Security and Protection in Information Processing Systems*, pages 473–487, 2004.

[20] H.M. Levy. *Capability-based computer systems*, volume 12. Digital Press, 1984.

[21] Myron Lipow. Number of faults per line of code. *IEEE Transactions on Software Engineering*, (4):437–439, 1982.

[22] C.K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Notices*, volume 40, pages 190–200, 2005.

[23] Spiros Mancoridis, Brian S Mitchell, Yihfarn Chen, and Emden R Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *ICSM'99*, pages 50–59. IEEE, 1999.

[24] S. McCamant and G. Morrisett. Evaluating sfi for a cisc architecture. In *USENIX Security*, page 15, 2006.

[25] G. Morrisett, G. Tan, J. Tassarotti, J.B. Tristan, and E. Gan. Rocksalt: better, faster, stronger sfi for the x86. In *PLDI*, pages 395–404, 2012.

[26] D.G. Murray and S. Hand. Privilege separation made easy: trusting small libraries not big processes. In *EuroSec*, pages 40–46, 2008.

[27] A. One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.

[28] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *USENIX Security*, 2003.

[29] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[30] R. Wahbe, S. Lucco, T.E. Anderson, and S.L. Graham. Efficient software-based fault isolation. In *SIGOPS OSR*, volume 27, pages 203–216, 1994.

[31] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. *Mobile Object Systems Towards the Programmable Internet*, pages 49–64, 1997.