Data Sandboxing: A Technique for Enforcing Confidentiality Policies

Tejas Khatiwala Raj Swaminathan V.N. Venkatakrishnan

Department of Computer Science
University of Illinois, Chicago
{tkhatiwa,rswamina,venkat}@cs.uic.edu

Abstract

When an application reads private / sensitive information and subsequently communicates on an output channel such as a public file or a network connection, how can we ensure that the data written is free of private information? In this paper, we address this question in a practical setting through the use of a technique that we call "data sandboxing" . Essentially, data sandboxing is implemented using the popular technique of system call interposition to mediate output channels used by a program. To distinguish between private and public data, the program is partitioned into two: one that contains all the instructions that handle sensitive data and the other containing the rest of the instructions. This partitioning is performed based on techniques from program slicing. When run together, these two programs collectively replace the original program. To address confidentiality, these programs are sandboxed with different system call interposition based policies. We discuss the design and implementation of a tool that enforces confidentiality policies on C programs using this technique. We also report our experiences in using our tool over several programs that handle confidential data.

1 Introduction

Sandboxing (or run-time monitoring) is a powerful and practical technique to protect an application that receives input from external sources. Often, these sources are not trustworthy and may send input that may victimize these programs. By monitoring the code for adherence to a specific policy, sandboxing prevents these attacks from being successful.

The two most common methods of implementing sandboxing are *in-line reference monitoring* (where code that implements the monitor is embedded in the same application) and *interface interposition* (where the monitor code is implemented to run at a specific interface boundary). Inline reference monitoring can technically enforce a richer set of policies when compared to interface interposition, as policies for inline monitors can specify constraints about internal variables of a program. An example of such a policy constraint is "the variable `numfiles` representing the number of files currently opened by a program never becomes more than 50".

For C programs, system call interposition, an instance of interface interposition, is the preferred approach for the main reason that the technique is *non-bypassable*. Programs that receive untrusted input can be subjected to running arbitrary code due to the absence of strong memory protection in the C language. Such arbitrary code can bypass the checks placed in the inline reference monitor. Given the fact that the operating-system kernel / application boundary can be monitored and security-related actions of programs must be ultimately effected via system calls, system call interposition does not usually suffer from the non-bypassability problem when implemented correctly.

System call interposition has therefore enjoyed a lot of attention as a practical technique in several earlier works [8, 1, 13, 17, 14, 7]. Mechanisms for implementing policies based on system calls is also a well-studied topic. For instance, the policy listed in the above example can be tracked using a finite-automata based monitor that tracks whether "the number of active `open` calls made by the program is less than 50".

The simplicity of policy enforcement in system call interposition comes at a cost. Monitors written for this approach cannot reason about the internal data-flow in a program, since they operate exclusively at the application/OS kernel boundary. Consequently, they tend to be coarse-grained and cannot enforce policies such as confidentiality of private data whose enforcement depends on the internal data flow in the program.

To illustrate this further, let us consider a confidentiality policy for a program that plays music files. It is not unusual to find that such programs periodically connect to an external server, possibly to obtain commercial information or update information. The user considers her music files and preferences sensitive and therefore these are not to be sent to the server. To prevent sensitive information from escaping the program, the sandboxing policy over system calls will need to prevent any write operations to public output channels such as files and network connections. This policy will definitely prevent information from leaking, but is very likely to trigger a loss of functionality with respect to these output operations, even if they involve non-sensitive data.

The new approach presented in this paper, called *data sandboxing*, combines the relative merits of inline reference monitoring (ability to incorporate data-flow information in policies) with the practicality of system call interposition (i.e., non-bypassability and easier policy development). The high level idea used in the approach is similar to *program slicing* [20]. A slice consists of instructions of a program that are affected by a set of variables from some program point, known as the slicing criterion. (We note that our approach always proceeds in the forward direction, starting from the criterion, and our notion of slices is slightly different from the standard notion, as explained in the next section). We identify instructions of the program that act on sensitive information. Based on this analysis, we *partition* the original program to meet the requirements of the confidentiality policy.

The application of this partitioning technique to the enforcement of a broad class of *information flow confidentiality* policies is discussed in this paper. These policies prevent private information from leaking from the program. Our technical presentation is focused on enforcement of these policies. Enforcement of other policies that gain precision by using data-flow information in programs is very much possible using our approach. However, we do not discuss them further in this paper. In addition, confidentiality policies also include lattice based multi-level policies [2], but our focus here is on dealing with only two levels: "sensitive" (high) or "public" (low). We also do not consider the effects of *covert channels* such as implicit flows, timing or storage channels [16]. As remarked in [24], additional research in this field is needed to make these techniques applicable to a wide range of programs.

While partitioning can be done manually for each program, it is not a viable option, considering the sake of both correctness and usability. Therefore, we have created a tool that partitions C programs in an automatic manner. With the help of our tool, we have been able to successfully enforce such policies on several applications. Using our approach, users who are programmers can freely modify programs running in their systems in order to guarantee confidentiality. End-users can freely benefit from these changes.

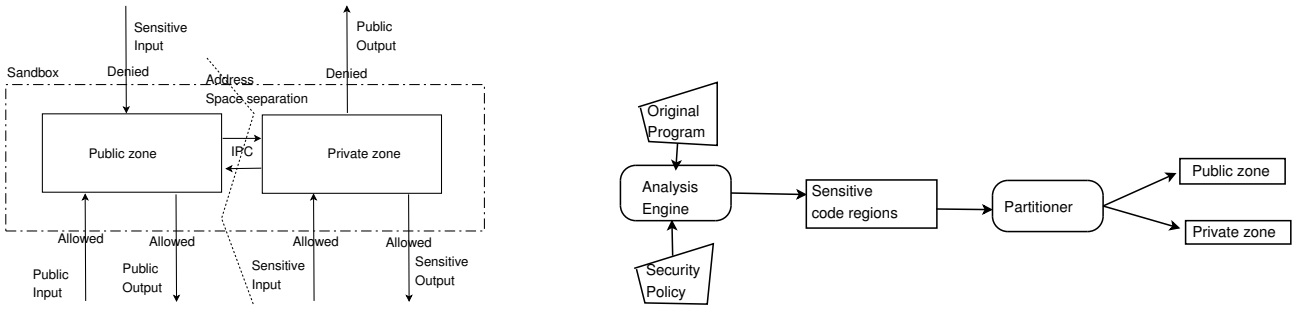
Creating separate programs for security reasons may seem to be a heavyweight solution, but is not new to the security community. The idea of privilege separation is based on creating separate programs to provide security assurances, as illustrated by the design of qmail [3] (a very widely used mail transfer agent), and in partitioning of programs such as OpenSSH [15, 4]. These projects also illustrate that security assurances outweigh any concerns due to overheads imposed by the partitioning solution. Our approach uses such partitioning techniques for enforcement of confidentiality policies.

This paper is organized as follows: Section 2 gives a technical overview of the approach and the various components of our system. Section 3 describes the design and implementation of our core analysis engine. The results of the analysis are used in the partitioning procedure described in Section 4. Section 5 describes our experience in using the tool for various free/open source utilities. Section 6 discusses related work. In section 7 we conclude.

2 Approach overview

Our approach involves splitting the original program into two partitions. We call these partitions *public zone* and *private zone*. The runtime view of these new programs is shown in Figure 1 (a). The private zone is the set of instructions of the original program that operate on potentially sensitive data. The public zone consists of all other instructions. The original program execution is substituted with the execution of these two programs. Whenever the public zone needs to read or process sensitive data, it initiates a *domain transfer* operation, which involves some IPC operations, resulting in a logical transfer of control to the private zone. These communication operations do not involve sending or receiving sensitive data. The private zone proceeds to execute code that processes sensitive input. Once operations on sensitive objects are done, or when non-sensitive operations need to be performed, a domain transfer is performed again to the public zone. At any further point in the program, if access to sensitive information is required, another pair of domain transfers happen.

As noted in the figure, the public zone is disallowed from reading sensitive input, while the private zone



(a) Runtime view

(b) Static analysis and slicing

Figure 1. Approach overview

is disallowed from writing to public output channels. Therefore, the private zone is the only component that receives, processes and stores sensitive data handled by the program. It thereby retains sensitive data completely within its address space.

The partitioning of the program into two components provides the following two main benefits:

- *Controlling sensitive data availability.* By our design objective, the private zone only runs code that is needed to process sensitive information. Sensitive data is shielded from the public zone, as it runs in a different address space. Therefore, any bugs (such as buffer overflows) in other routines of the public zone (i.e., when performing other non-sensitive program tasks) will not result in unauthorized disclosure of sensitive information. For example, this approach will prevent an attack on an ssh server that targets to steal the server's master secret (its private key).
- *Appropriate confinement for different zones.* The private zone is automatically confined such that it cannot write to public locations, while the public zone is disallowed from reading any sensitive channels. This is impossible to achieve in the case of the original (monolithic) program without breaking it or losing some functionality. In our music player example given above, the operations related to reading (sensitive) music preferences are performed in the private zone, while network operations (that involve sending and receiving non-sensitive data) are performed in the public zone. This allows the music player example to be successfully used by the user, while ensuring that user's privacy is never violated.

2.1 System overview

Figure 1(b) shows the basic architecture of our approach. A security policy is provided along with the program to an *analyzer*. The analyzer performs data flow analysis and deduces a list of variables (for every program point) that may potentially contain sensitive values. It then performs a second analysis where it identifies the set of sensitive instruction regions in the program. These regions are then taken by the *partitioner* to create the public zone and private zone with appropriate transfer routines that enable the domain transfer operation between these programs. The resulting programs are confined using system call interposition policies at runtime. We describe these operations in the following sections.

2.2 Security Policy Specifications

In our approach, security policy specifications specify the confidentiality requirements on the original program. There are two parts to a policy specification: a *runtime* part that is used to confine the program during its execution, and a *static* part that is used by our tool to analyze and partition the program.

Runtime confinement policy These are policy specifications over the alphabet of system calls that are enforced on the application when it is executed. Separate policies are written for the public zone and the private zone respectively to meet our confinement objectives. Specifying and enforcing such policies is fairly standard, as illustrated in several past works such as [8] and [18]. We do not discuss them any further in this paper.

Policy specification for static analysis Static policy specifications are used by our analyzer to infer the flow of sensitive information in the program. These are written based on function prototypes. Let us say `int f(a,b)` is a function, where the parameters `a` and `b` are passed by value. Then the policy specification for `f` may be stated as follows: `f(high, -) --> high`

and `f(low, -) --> low`. This means that the return value of `f` is sensitive if and only `a` is sensitive. Similar specifications are provided for values provided by reference as they act as outputs. Such input output specifications are used in the following cases:

- *Sensitive input routine specifications* User input enters a program using a function call or a system call that reads from an input channel. For instance, in the `su` program, the function `getpass` reads user input. For this function, while the input is not sensitive, the output is. The analyzer requires these inputs for propagation of sensitive values further in the program. These are specified using the same notation that is used for system calls that is described below.
- *System call specifications* System calls are not analyzable, and hence our approach requires us to specify the input/output semantics of system calls. For instance, a `int read(int fd, void *buf, size_t count)` system call that has the following policy specification: `low read(high, low/high, low)`. It suggests that if the first argument (the file descriptor) is a sensitive value, then the second argument `buf` (passed by reference) points to a sensitive (`high`) value at the end of the read operation. Similar specifications are added for external library calls that are not analyzed by our tool. Note that we do not require specifications for all system or library calls, but for only those that are used to process sensitive information.

Occasionally, the policy may need to refer to the input program line numbers to annotate the sensitive input processing routines. A typical example is an `open` system call whose file name argument is not statically available. In this case, the programmer needs to provide additional location information of the call in the source file as part of the policy.

```
/*  .. ..*/
1. int pin, cpin, flag;
2. pin = getinput();
3. cpin = crypt(pin);
4. fd1 = open(SECRET-FILE,.. );
5. fd2 = open(LOG-FILE,..);
6. stored-pin= read-from-file(fd1,..);
7. if (cpin == stored-pin) flag = 1;
8. else          flag = 0;
9. write(fd2,..);
/*  .. ..*/
```

We use the above toy example as a running illustration of the ideas used in this paper. It is a simple authentication routine of a program that accepts a 32 bit PIN number as input, compares it to the stored PIN and writes the result of authentication to a public log file.

The policy specifies that the input function `getinput` returns sensitive information, and `SECRET-FILE` is sensitive (high), and `LOG-FILE` is public (low). A correct enforcement of this policy will prevent the PIN information from going to the log file.

3 Analysis

The objective of the analysis engine is to identify instruction regions of the program that handle sensitive information. Once this analysis is done, the set of sensitive instruction regions is given to the code generation step that partitions the code into public zone and private zone. As mentioned earlier, we ignore the effect of implicit flows [16] and any covert channels arising out of timing or storage channels. Also, we wish to point out the distinction here between the objectives of our analysis module and those that use static analysis for information flow. The objective of the analysis engine is not policy enforcement (i.e., ensuring that sensitive data does not get written to public output channels). As already noted, that such analysis for enforcement is not reliable in the context of C programs and we require sandboxing. The purpose of the analysis is to *merely* identify the set of sensitive instruction regions of the program.

3.1 Basic steps

The analysis proceeds with the following steps:

1. Propagate sensitive values across the program and identify all potential variables that may receive sensitive information.
2. Identify the set of instructions that act on these variables.
3. Logically group these instructions into sensitive instruction regions to be executed in the private zone.

The analyzer begins by using policy information to identify the *context function*, which is a function in the program which make use of the sensitive input routines. There can be many such functions, but for the purposes of this discussion we limit ourselves to one function. We start with a set of sensitive variables that is initialized from the policy. For each program point p in the context function, the analyzer computes a set V_s^p of sensitive variables. This computation is performed for each location based on the semantics of the instruction. This is done using standard information flow rules for various program constructs. If a branch is encountered, the analyzer maintains and updates independent V_s^p sets for each point p along each path, thus maintaining path sensitivity. When these paths merge, the union of these sets

is used for the merge point. At a call instruction, the analyzer examines the called function. If the definition is available, it analyzes the function and computes the sensitivities of any arguments passed by reference and return values. In the absence of a definition or a policy specification for a particular routine (i.e., an external function) and if any argument to the function is sensitive, then the analyzer considers the return value and any arguments passed by reference as sensitive.

For the example program, based on the input policy and analysis, variables `pin`, `cpin`, `fd1`, `stored-pin` are sensitive at various locations. Therefore, the values held in these variables at these locations need to be protected.

The next step is to identify the set of instructions that handle sensitive variables. (In our implementation, these steps are done concurrently.) If an instruction or a statement at location p uses a variable that belongs $V_s{}^p$, that instruction is *marked*. In our example program, instructions in lines 2, 3, 4, 6, 7 and 8 are marked. If the marked instruction is a function call then the corresponding function definition is also included in the private zone. A transitive closure of the set of such callees is computed and all the definitions in this set are marked.

Block creation One way to partition code that handles sensitive data, as opposed to the above marking approach, is to use a conventional forward slicing technique. While this approach is likely to yield programs that are self-contained, they will include instructions that may exclusively deal with non-sensitive variables. This may eventually lead to a bloat in the size of the child program. We have instead chosen to identify regions of instructions that exclusively handle sensitive data.

The analyzer gathers these sets of sensitive instructions to identify sensitive code regions or blocks. A sensitive code region comprises of one or more sequential marked instruction(s) / statement(s). If control enters the first instruction of the region, it exits only through the last instruction. There can be several such regions. When the analyzer encounters the first marked instruction, it begins a new block. Every consecutive marked instruction is added to the block. The first non-marked instruction encountered marks the completion of the current sensitive code block. Any intervening unmarked instruction would therefore result in at least two blocks. A branch is usually considered the start of a new block unless the entire branch statement itself is marked.

For our example, two blocks are created. The first block contains instructions in lines 2, 3 and 4 and the second one contains line 6, 7 and 8. Therefore, the operation that opens the (public) `LOG-FILE` will be performed in the public zone.

Analysis for state exchange between zones While blocks provide the needed abstraction, it is also necessary to maintain program state across the public zone and private zone. The analyzer uses a structure V_B to keep track of the variables that are used in the instructions of the block B. This information is necessary to generate code that provides the private zone with these runtime values. If V_{start} is the set of sensitive variables at the start of the block, then the set difference of V_B and V_{start} is the set of values that needs to be communicated at runtime.

When a block finishes execution, the program state has to be communicated back to the public zone. By definition of the private zone, note that no sensitive values are needed by the public zone. Hence, only values of updated non-sensitive values need to be returned back to the public zone. If V_{end} is set of sensitive variables at the end of a block, then the set difference of V_B and V_{end} is the set of values that need to be returned to the parent.

Declassification Cryptographic one-way functions such as $crypt$ are a special case. They do take sensitive input, and return output that cannot be used to recover the input. Any program analyzer cannot automatically deduce this to be the case. Such automatic inference can be shown to be undecidable by establishing a reduction from the halting problem. In this situation, the analyzer needs to know the input/output relationships of these functions to downgrade the output value, thereby performing declassification [16]. Another case that the analysis handles automatically is the assignment of constant values to variables. In this case, a sensitive variable becomes non-sensitive. Declassification and constant assignment control the growth of the V_s set of sensitive variables in our approach.

Use of arrays / structures Currently, an entire array or structure is considered sensitive or non-sensitive as a whole object.

4 Partitioning

Generation of private zone Once the set of sensitive blocks is analyzed, the programs corresponding to the public zone and the private zone need to be created. One intuitive way to generate the private zone is to create wrapper functions that enclose the sensitive instruction regions. Whenever a domain transfer is initiated by the public zone, these functions can be invoked by passing them all the non-sensitive values from the public zone.

A problem arises with such an approach concerning

how sensitive variables are shared between these wrapper functions. Again, there are two options in this case: a) have them in a scope that is global to all these functions or b) restore these values as a first step in every wrapper function, and save them on exit. However, the first option has a disadvantage of making the sensitive variables accessible to many other functions that don't require them. The second option has an obvious performance penalty. Therefore, the approach we have taken is to create blocks within the function itself. The sensitive variables that are shared across blocks can now be at the function scope. We use the facility offered by the CIL [10] tool that lifts all local declarations (including block-level) to the function scope level. The private zone is thus constructed with several blocks that are present in the context function, along with the corresponding local and global variables required by these blocks.

Generation of public zone The public zone program includes all the statements in the original program, with the exception of the sensitive blocks. This program is created with instructions to fork the private zone program and set up a communication channel. Note that, such instructions must be inserted before any sensitive information enters the original program. In the public zone, these instructions must precede any node where a domain transfer operation is initiated to the private zone. This can be achieved by placing these operations in a *dominator* node. The first instruction in the function is clearly a dominator where we have placed these instructions. Similarly the operation for waiting for the private zone needs to be done on a post-dominator node. For functions with multiple return program points, the one-return transformation [10] is a way of getting the post-dominator point. The new public zone program now replaces the original program executable.

Generation of checks that makes use of runtime information. Whenever the private zone is not processing sensitive information, it blocks on the communication channel for the public zone to initiate the domain transfer operation. It needs to know the right block to execute when it receives a message. However, this information is not statically available, as the parent can initiate a domain transfer from several possible locations, hence we cannot generate static target labels for the private zone. To solve this problem, we generate code that (at runtime) determines the target block to be executed. The public zone supplies this information at runtime, as it knows the block to be executed. The private zone on receiving this message finds the right block to execute. Lines 5 and 6 in the transformed example show the generated code.

```
1. int pin, cpin, flag;
2. START:
3.  msg = read_msg_from_parent();
4.  if (msg == TERM) exit(0);
5.  else if(msg == BLK1) goto Block1;
6.  else if(msg == BLK2) goto Block2;
7. Block1:
8.  read_n_demarshal(cpin, pin);
9.  pin = getinput();
10. cpin = crypt(pin);
11. fd1 = open(SECRET-FILE,.. );
13. goto START;
14.Block2:
15. read_n_demarshal(flag);
16. stored-pin= read-from-file(fd1,..);
17. if (cpin == stored-pin) flag = 1;
18. else         flag = 0;
19. marshal_n_write(flag);
20. goto START;
```

Serialization The exchange of program state information between the public zone and private zone is done by serializing (also known as marshalling) the program state into a byte array at that program point, and transferring the result to the private zone. In the private zone, this byte array is de-serialized to reconstruct the information about the environment. We have achieved automatic serialization of a program's state by generating stub modules for each data type that is exchanged during the domain transfer.

In our approach, complex data types such as structures and pointers are described using a messaging protocol that describes the individual data types and their type structure. The message passing is essential for values that can not be exchanged. For example, a pointer variable pointing to NULL can not be directly communicated through its value. Similarly non-null pointer references in public zone can be flattened but cannot be directly used in private zone process. For this reason, communicating pointer variables requires special treatment. To maintain a pointer variable's state consistent during domain transfer, we keep a table in each zone that maintains a mapping of the pointer's address in the other zone.

This procedure is adequate for most recursive data types such as linked lists and trees that we have tried in our examples. Our construction works correctly in these cases and we have manually verified them. However, more complex data structures may need manual intervention as they may have specific invariant requirements that cannot be deduced automatically.

Limitations While we believe our approach is general enough to be applicable to a large class of programs, our

implementation has two main limitations. Both of them relate to how pointers and aliasing are used in the target program. The first limitation concerns the use of aliasing. Currently our approach treats pointers as data references, treating any operations that directly make use of pointers as references to values. They will be correctly marked as sensitive. However, the presence of aliasing (where an object is referred to both by its original name and its alias) may invalidate this assumption. Whenever the analysis encounters the possibility of aliasing, it prompts the user for manual verification. In this case, we had to perform manual verification of the code generated for the use of aliasing. Our assumption is valid for the programs we have tested our tool with. In these examples, pointers were indeed used as references and hence our code generation approach was correct. Integration of points-to-alias analysis algorithms in our approach will ensure that we correctly identify all sensitive instructions and avoid such manual verification.

The second assumption concerns the size of objects referred by pointers. While serializing data to communicate the public zone state to the private zone, there are some situations where the sizes of objects pointed to by pointer data types is not known. In general, tracking pointer sizes is hard. One solution is to have fat pointers that record additional information, such as their sizes. Another viable solution is to have a shared memory region between the public zone and private zone that contains public data, and taking additional measures to protect sensitive data in the private zone. This will be explored as part of our future work. For now, the code generation uses some simple heuristics that make certain guesses about the nature of the variable being pointed to. For example, to serialize a variable of type pointer to `char`, we use the a runtime sample value of the strlen() function for the size of the variable. Another current option we had at hand was to have the user of the tool explicitly specify the size of the pointed-to variables. We preferred the first option even though it is a potentially unsound heuristic, as it seems to be a good tradeoff to the complexity of specifying individual object sizes. We adopted this approach, and followed up by manual verification for checking pointer sizes. Since the sizes of code blocks was small (see Figure 3), this was easily possible.

5 Evaluation

In this section, we describe the results with our prototype implementation. Our implementation uses the CIL [10] framework. We have chosen representative examples from several classes of open source programs: simple authentication programs, system monitoring and analysis programs, music related utilities that handle sensitive information. Our evaluation section is divided into three parts 1) policy enforcement evaluation, where we present the effectiveness of policies in preventing information flow 2) performance evaluation and 3) security analysis of the approach.

5.1 Policy enforcement evaluation

Linux-Monitor is a utility that polls system resources such as processes and disk partitions at specific intervals of time. It then proceeds to log this information into system logs or at a remote-server running on port 8881 (syslogd substitute for remote logging). Here we enforce a policy that prevents linux-monitor from logging to a remote connection when it reads from system resources considered sensitive by the system administrator. The policy enforcement successfully prevented this information from being communicated.

Htpasswd is used to create and update flat-files, that store user names and password for basic authentication of HTTP users. It calls the $crypt$ routine and then writes encrypted passwords to a user specified file. Our tool separates the code handling password information and enforces a policy that prevents this information from being written to any other file.

Mediachat is our own implementation that simulates a media player that allows users to connect to a chat server while reading and playing music files from their local disk. It is an example of an application that operates on sensitive (music preferences) and non-sensitive data (network chat) at the same time. The policy is to prevent the list of music files from being communicated on the network to the server. A typical sandboxing policy would have prevented the chat application from correctly functioning. Our data sandboxing approach allowed the chat to proceed while preventing any private music information from being sent to the network.

chfn, chsh, passwd These are standard Unix administrative utilities. Our tools successfully partitioned the authentication code in these programs. We sandboxed the private zone programs to ensure that sensitive information is only written to the system password files (in the case of passwd, chfn and chsh). The sandbox for the public zone programs need a setuid wrapper, as these are setuid programs and the ptrace mechanism we use for user-level system call interposition does not cross setuid boundaries. A kernel interception mechanism can handle this situation.

5.2 Evaluation

The performance of the system was analyzed both for the overall performance and for the domain transfer operation using micro benchmarks. We ran experiments on a Intel P4 3.4 GHz processor with 2 GB of RAM running linux kernel version 2.6.9.

Micro benchmark on the domain transfer operation We measured the base overhead for a domain transfer between public zone and private zone. This overhead results from setting up a communication channel and performing serialization. Of the programs we tested, the linux-mon program had the worst case domain transfer overhead. We monitored five different resources and the time to serialize shared data was 2.86 milliseconds. The performance penalty for domain transfer operation (cost of communication setup + serialization and deserialization as opposed to simply executing instructions in the original program) was a factor of 6.16.

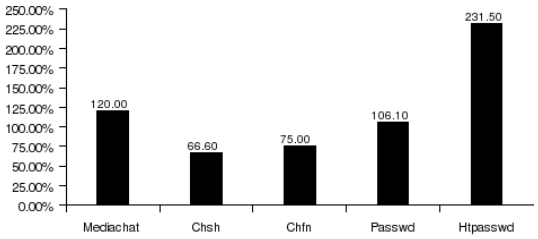


Figure 2. Performance Overheads

Overall performance measurements We measured the overall performance (public zone + private zone, combined user + system time, without system call interposition) of the transformed programs compared to the original programs, and the results are shown in Figure 2. Linux monitor is a continuously running server program that keeps reading sensitive files and so we could not use this form of comparison. From the results, we note that the worst overhead is about 231.5% for htpasswd, and is primarily due to the new address space creation. System call interposition techniques at the kernel level would add about 10-15% overheads.

Performance Improvements Our implementation is currently not optimized for performance. The major factors contributing to the overheads are the domain transfer operation and the associated copying of state. Several optimizations are possible. The current implementation uses address spaces for memory isolation. An intra-address space protection mechanism such as software fault isolation [22] can result in much lower overheads. Also compiler optimizations such as code motion can be used to produce fewer state transfers, and live variable analysis may result in lowering copying overheads.

Program Name	Orig. (# LOC)	Marked ins (# LOC)	# of Blocks	Avg. # LOC/block
passwd	2333	193 (8%)	6	4
htpasswd	984	240 (24%)	4	12
chfn	1238	126 (10%)	1	18
chsh	1138	112 (10%)	1	18
Mediachat	335	92 (27%)	2	7

Figure 3. Program size information

Additional memory overheads Since our approach creates a new private zone process for every program, there is an additional memory overhead due to the process creation. We chose to measure memory overhead for to the `htpasswd` program, as the private zone process in this case has the worst case memory consumption both for code as well as data. We inserted a break point in the code of the private zone, and measured the resident memory size of the process. The additional memory required was 2.5 Megabytes. Other programs listed above tested will consume significantly less memory than `htpasswd`.

Program sizes Figure 3 gives a table of the analyzed program sizes in LOC, the number of marked (sensitive) instructions in LOC, the number of blocks and the average block sizes respectively. The actual sizes of the programs are much larger, the LOC was measured without counting the code from libraries. Also the sizes of the private zone programs are the sizes of these marked instructions plus a linked library of around 200 lines of C code that performs serialization. We observe that the percentage of instructions that handle sensitive data in these programs ranges from 8% to 27%. The highest is for Mediachat, which performs a significant fraction of its code in handling sensitive (music) data.

5.3 Security analysis of the approach

Our approach also prevents damage from attacks that subvert the program to gain access to confidential data. In this case, the (minimal) code that runs in the private zone only reads and processes sensitive information. The public zone if subverted, thus cannot be directed to read sensitive information. Secondly, the code that runs in the private zone does not share sensitive data with the public zone. Also, the communication channel between the public zone and private zone is a pipe, which is only available to these two processes and hence cannot be observed by any local processes acting on behalf of the attacker. It is possible that the private

zone can have code (derived from the original program) to have bugs that can be exploited. In this case, we first note that our approach does not do any worse than before, as the original program still can be exploited. Secondly, since the private zone runs the minimal piece of code required to run sensitive information, as opposed to the public zone which runs code that does the bulk of the work of the program, chances of such bugs being present in the private zone is reduced. Thus the code that reads confidential data is separate from the remaining processing code, and this mitigates security risks of losing confidential information.

6 Related work

Sandboxing Sandboxing based approaches [8, 1, 13, 17, 14] involve observing a program's behavior and blocking actions that may compromise the system's security. This is achieved in most of these works through system call interposition. This approach is transparent to the program, and policy enforcement comes without needing to modify programs. However, the monitor cannot distinguish between sensitive and public data that is written in an output channel. Consequently, sandboxing approaches do not allow program actions that write to public output channels, and could therefore be limiting application behavior in many situations. In this work, we have provided an approach that addresses this problem.

Static analysis Several static analysis based approaches [5, 6, 19, 9] have been successfully used for finding bugs in C programs. However, static analysis alone cannot be used reliably in policy enforcement for C programs, as programs can be compromised due to lack of memory safety. Though our approach uses static analysis, this is merely done to partition a program. Our enforcement technique is based on monitoring at the application/kernel boundary, and hence is not bypassable.

Information flow analysis There is a long history of work in information flow analysis of program. Work on information flow started with the work of Bell and LaPadula [2] in the context of processes in an operating system. More recent work [21, 11] brought this work in terms of data flow in programs. Most of these approaches rely on type safety for enforcement of policies. In a language such as C which is not type safe, direct application of such policies will not result in reliable enforcement. In such systems, confidential data continues to remain in the program's memory once a program is victimized, and is readily available without any protection. The goal of this work was to minimize the window of disclosure of confidential information. Though we use static analysis for inferring possible data flows, our enforcement uses the combination of address space separation of private data and sandboxing for reliable enforcement of such policies.

Taint based approaches Taint based approaches [12, 23] have been recently used in detection of data-oriented attacks such as injection attacks. Taint-based approaches do have the scope for providing finer granularity to assist sandboxing. In all the above works, taint based enforcement has been used for protecting system integrity, i.e., ensuring that data from untrusted sources do not compromise trusted destinations such as shell commands. In this work, we presented an approach to solve the dual problem of confidentiality, where input from trusted sources do not reach untrusted (public) output channels.

Program partitioning approaches A program transformation technique *Jif-split* was described in [25] in the context of distributed programs running on untrusted hosts. Programs written in a language called *Jif* are annotated with security types, and the system automatically splits the program to match the enforced security policy described through these types. There are several differences with respect to our approach: First, their approach is in enforcement of distributed programs such as web services in the context of untrusted hosts. Our approach is for protecting the confidentiality of inputs in trusted hosts that may receive untrusted inputs. Secondly, their approach requires writing programs in a special type-safe language for Jif to accomplish the partitioning, and our approach is focused on retrofitting C programs that handle confidential information for protection of sensitive data.

Privilege separation is another idea that uses separation of program on the basis of privileges required to minimize security risks. It was used in the design of programs such as qmail [3], and later in retrofitting of programs such as OpenSSH manually [15] and automatically [4]. While the goal of these approaches is to separate code running with special privileges, our approach is focused on using using a similar approach to make private data inaccessible to the parts of the program that do not require it. *Privtrans* [4] was the first work that applied an automated technique for privilege separation in the realm of C programs. We follow a similar partitioning approach that partitions programs based on process-level separation. However, there are two main differences. Their approach to partitioning unprivileged and privileged code is based on a delegation / authorization model, where the unprivileged code requests the privileged code to perform certain operations. This is suit-

able for privileged operations such as setuid, where explicit operating system permissions are required to perform these operations. However, it may not be suitable for operations that handle private data, as the unprivileged program can be compromised to perform such an operation (say reading a sensitive file) by itself without needing special permissions or having to request the privileged program to perform that operation. (For instance, this can be done after a compromise through a *return-into-libc* attack.) In our approach, such an operation is prevented through the policy enforcement by system call interposition. Secondly, they assume the procedure level abstraction as the boundary for partitioning. This method of partitioning is unsuitable for our approach that intends to minimize the instructions that can access sensitive data, and thereby making it inaccessible to portions of the program that do not require it. Doing this requires us to insert domain transfer code in the body of functions and not necessarily at function boundaries.

7 Conclusion

In this paper, we presented an approach called data sandboxing for protecting confidential information from unauthorized disclosure. Our approach works by partitioning program instructions into two parts, the public zone and private zone. These two parts are isolated from each other through address space separation, thus making the sensitive data in private zone inaccessible to the public zone part of the program. Our approach has potential to prevent sensitive information from disclosure in the event of a compromise. It also overcomes the disadvantages of pure sandboxing approaches which prevent external communication in order to enforce confidentiality. We illustrated the use of our approach by applying it to several examples.

References

[1] A. Acharya and M. Raje. Mapbox: Using parameterized behavior classes to confine applications. In *USENIX Security Symposium*, 2000.

[2] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, Vol. 1, MITRE Corp., Bedford, MA, 1973.

[3] D. Bernstein. The qmail mail transport agent. Available from `http://cr.yp.to/qmail.html`.

[4] D. Brumley and D. Song. Privtrans: Automatic privilege separation. In *USENIX Security Symposium*, San Diego, 2004.

[5] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *ACM Conference on Computer and Communication Security (CCS)*, 2002.

[6] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2000.

[7] T. Garfinkel. Traps and pitfalls: Practical problems in in system call interposition based security tools. In *NDSS*, 2003.

[8] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications: confining the wily hacker. In *USENIX Security Symposium*, 1996.

[9] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *USENIX Security Symposium*, Washington, D.C., August 2001.

[10] S. McPeak, G. C. Necula, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for C program analysis and transformation. In *Conference on Compiler Construction*, 2002.

[11] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering Methodology*, 1999.

[12] J. Newsome and D. Song. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, 2005.

[13] V. Prevelakis and D. Spinellis. Sandboxing applications. In *Usenix Technical Conference Proceedings: FREENIX Track*, pages 119–126, Berkeley, CA, 2001.

[14] N. Provos. Improving host security with system call policies. In *12th USENIX Security Symposium*, 2003.

[15] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *USENIX Security Symposium*, Washington, D.C., August 2003.

[16] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1), Jan. 2003.

[17] K. Scott and J. Davidson. Safe virtual execution using software dynamic translation. In *18th Annual Computer Security Applications Conference*, Las Vegas, Nevada, 2002.

[18] R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney. Model carrying code: a practical approach for safe execution of untrusted applications. In *ACM Symposium on Operating System Principles*, Bolton Landing, New York, October 2003.

[19] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *USENIX Security Symposium*, Washington, D.C., August 2001.

[20] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[21] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[22] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *proceedings of the Symposium of Operating System Principles*, 1993.

[23] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security Symposium*, 2006.

[24] S. Zdancewic. Challenges for information-flow security. In *1st International Workshop on the Programming Language Interference and Dependence*, 2004.

[25] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *ACM Symposium on Operating System Principles*, 2001.