Detecting illegal system calls using a data-oriented detection model

Jonathan-Christofer Demay[1], Frédéric Majorczyk[2], Eric Totel[1], and Frédéric Tronel[1]

[1] Supelec, Rennes, France, first_name.last_name@supelec.fr
[2] IRISA / Université de Rennes 1, Rennes, France, first_name.last_name@irisa.fr

Abstract. The most common anomaly detection mechanisms at application level consist in detecting a deviation of the control-flow of a program. A popular method to detect such anomaly is the use of application sequences of system calls. However, such methods do not detect mimicry attacks or attacks against the integrity of the system call parameters. To enhance such detection mechanisms, we propose an approach to detect in the application the corruption of data items that have an influence on the system calls. This approach consists in building automatically a data-oriented behaviour model of an application by static analysis of its source code. The proposed approach is illustrated on various examples, and an injection method is experimented to obtain an approximation of the detection coverage of the generated mechanisms.

1 Introduction

Generally speaking, an attack against an application consists in exploiting a vulnerability in order to violate the confidentiality or integrity properties of the system or the application under attack. In the context of intrusion detection methods at application level, a lot of existing work focuses on the detection of the violation of the integrity property. Attacks against a process can consist in corrupting either the control-flow of the program (e.g., to execute injected code), or the data items manipulated by the program during its execution. A lot of papers focus on the detection of the program control-flow corruption, either considering the process as a white box, or seeing it as a black box. An exemple of a white box approach is to verify during the execution that the control-flow graph of the program is legal. An example of the black box approach consists in verifying that the trace of the process execution in the system is correct (e.g., the sequence of system calls [1]). Both approaches can be subject to false negatives, as the attacker can either corrupt data items that do not influence the control-flow of the program, or perform attacks that mimic [2] the normal behaviour of the application. Various papers [3,4,5,6] have enhanced the black box approach in order to detect these types of refined attacks.

In this paper, we propose a white box approach for intrusion detection that aims at detecting the corruption of the data items in an application, so as to detect erroneous system calls (e.g., their arguments are not correct, or the data that led to their execution were incorrect). The approach relies on the building of a data-oriented behaviour model. This method can be presented as an interesting complement to the usual control-flow

corruption detection method, in order to detect data oriented attacks. To attain this goal we use static analysis to build constraints on intrusion sensitive data items, then we instrument the software with executable assertions that check these constraints during the execution of the program.

The contribution of this paper is not to provide new static analysis techniques, as our work relies on an off-the-shelf static analyser called Frama-C [7]. However, we want to show on real-life examples that a detection model can be built by static analysis and detect data attacks (even unknown ones).

The paper is organized in the following way: after a short related work section on white-box attack detection, we show how to build the behaviour model and emphasize the accuracy of the model on a previously known attack. Then we show the results of the software instrumentation on various examples. At the end we evaluate on an example the detection rate we can expect from the generated detection mechanisms.

2 Related Work

We believe that white box mechanisms can help improving the detection performance as they are able to take advantage of the internal state of the monitored program. Indeed, they have access to all the internal data structures and algorithms used by the program.

That is the case, for example, with Control-Flow Integrity [8] and Program Shepherding [9]. These generic techniques verify the integrity of the control-flow of a program. A control-flow graph of the program is computed prior to its execution and then used at run time to check the integrity of the process control-flow. Because mimicry attacks still need to force the program control-flow to deviate from valid execution paths, they are caught by these approaches. However, unlike our approach, all those techniques are completely ineffective against computation data attacks (also called non-control data attacks [10]), since these attacks are performed using a valid execution path.

Other white box approaches that focus on non-control-data attacks and that do not exhibit this weakness have been proposed. For example, Write-Integrity Testing [11] enforces control-flow and data-flow integrity in a program. In the work on Data Flow Integrity [12], a data-flow graph is computed prior to the execution. It contains, for each data item read by an instruction, the set of instructions that may have written its current value. This data-flow graph is then used at run time to verify the integrity of the data flow of the process. If the program has a vulnerability that is exploited to corrupt some data, the next time this data is read a deviation from the data-flow graph will be observed allowing thus the detection of the attack. This type of approach is very effective against all kinds of non-control-data attacks, but use a very different philosophy than our approach. They focus on the illegal modification of the data, whereas in our approach we focus on the correctness of the data. As a consequence some attacks missed by the data-flow integrity method (such as an illegal value stored in a correct variable) can be detected by our approach. Conversely some illegal writes can be missed by our approach (a legal value can be written in an incorrect variable), making both approaches complementary.

3 Intrusion Detection

In this section, we explain how non-control-data attacks are real threats and how a data-oriented behavior model can detect them. We also present *SIDAN* [3] (Software Instrumentation for the Detection of Attacks on Non-control-data) [13], a tool we have developed that implements our detection model.

3.1 An attack against non-control-data

```
00. int main(int argc, char ** argv){
01.     char buffer[256];
02.     uid_t uid = 5;
03.
04.
05.     seteuid(uid);
06.
07.     while(aux = fgets(buffer, 256, stdin))
08.    {
09.         seteuid(0);
10.         printf(buffer);
11.
12.         seteuid(uid);
13.    }
14. }
```

```
00. int main(int argc, char ** argv){
01.     char buffer[256];
02.     uid_t uid = 5;
03.
04.     assert(uid == 5);
05.     seteuid(uid);
06.
07.     while(aux = fgets(buffer, 256, stdin))
08.    {
09.         seteuid(0);
10.         printf(buffer);
11.         assert(uid == 5);
12.         seteuid(uid);
13.    }
14. }
```

Fig. 1. Example of string format vulnerability and useful assertions

Chen et al. [10] have demonstrated that non-control data attacks can be as severe as control-data attacks on various real world vulnerabilities. Among them, a vulnerability found in the implementation of the open source ftp server *wu_ftpd* will serve as an example to illustrate our approach. Figure 1 (left column) is an excerpt of the original code exhibiting the same vulnerability. Line 10, a string taken as user input (line 7) is printed without using a string format. Consequently, a user can forge an incorrect buffer containing string formats that allows to write directly in memory. In this case, the target could be the *uid* variable. As a consequence, the attacker can elevate its privilege at line 12, without corrupting the execution path, by forcing the parameter of the *seteuid* call to be the administrator identifier (zero). This example shows how such an attack violates a very simple constraint on the *uid* variable. Indeed, the *uid* variable should remain constant during the execution of the loop (lines 7 to 13) and should be equal to the value it has been assigned at *uid* (line 2). The problem we tackle in this paper is to automatically build such constraints in order to detect attacks at runtime.

3.2 Data oriented detection Model

In our approach, we consider that an attacker aims at modifying data items in the memory space of a process in order to execute one or more incorrect system calls. This objective can be fulfilled in two ways: either the attacker alters variables that influence

[3] http://www.rennes.supelec.fr/ren/rd/ssir/outils/sidan/

the internal control-flow of the program (and thus executes system calls in an incorrect context), or the attacker modifies directly or indirectly the values of the parameters of one or more system calls (and thus executes legal system calls with incorrect values). Both types of attacks aims at modifying non-control data items in the program. Note here that non-control data items are all variables used by the program source code, and can thus have an impact on the control-flow of the application. They are opposed to control-data items that are used by the system (and not the application) to control the execution flow of the application (e.g., a return adress on the stack).

In order to detect these modifications, we propose to identify the set of constraints that should be verified at runtime for these items. Generally speaking, these constraints can be divided in two classes: the variation domain of the variables (e.g., a variable can take a restricted set of values), and the relationship between the variation domains of the variables (i.e., when a variable has particular values, other variables take a defined set of values). If we only check if a variable is within its variation domain, it may be easy for an attacker to impose a reasonable value that would fit in the variation domain, but that is incorrect in the context of the program. Clearly, if we can maintain the relationship with other variables, it will be more difficult for an attacker to modify simultaneously several variables that depend on each other while keeping the program in a consistent state. As a consequence, we propose to define a data behaviour model for intrusion detection that aims at taking into account these requirements.

Formally, we define for a given system call SC_i its data behavior model by a triple (SC_i, V_i, C_i) where V_i is the set of variables the system call depends on, and C_i the set of constraints on these variables that can be deduced from the program analysis. We can define the normal data behavior model of the program by the set of all triples, $DBM = \{\forall i, (SC_i, V_i, C_i)\}$. In the following section, we address the two problems faced to build this model: how to determine the set of variables a system call depends on, and how to obtain the constraints that must be verified on these variables at runtime.

Building the set of variables Building V_i requires the ability to determine in the program which are the variables that influence the execution of the particular system call SC_i. Generally speaking, a system call can depend on a variable in two different ways: a variable either has an influence on the path in the program that leads to the execution of SC_i or influences the parameters of SC_i. These sets of variables can be built by using a static analysis technique called program slicing [14]. A program slice can be defined as the parts of a program that potentially affect the values computed at some point of interest of this program. In our case, we are looking for all the variables that influence a system call, and thus all variables that are in the program slice whose point of interest is the system call itself. In the static analysis field, the computation of a program slice is generally based on the computation of a program dependency graph (PDG) [15]. The PDG is a directed graph whose vertices correspond to statements and control predicates, and edges correspond to data and control dependencies. This graph can be used to exhibit the set of variables a particular system call depends on, and the type of dependency. In our implementation, we directly use the PDG notion to discover in the program all variables a system call depends on. To illustrate this paragraph, we can consider the example on Figure 1: the *seteuid* call at line 5 depends on one vari-

able: *uid*. However, the *seteuid* call at line 12 depends indirectly on the *aux* variable and directly on the *uid* variable.

Constraint discovery Automatically discovering constraints in the source code on the variables that are defined in the previous paragraph requires to use static analysis techniques. We could imagine any types of constraints, including for example temporal constraints. In practice, static analysis techniques often compute constant constraints, also called invariants. Indeed, any static analysis technique that is able to compute invariants from the source code fits our needs. Moreover, a popular technique for calculating such invariants is the abstract interpretation method [16]. In practice, abstract interpretation provides a way to find properties on the variables of a program by computing abstract domains that represent abstractions of the real properties of the program. Several models have been developed to discover such invariants. Among them, we have chosen to focus on the build of numerical abstract domains, i.e., we intend to find numerical invariants. These types of domains can be classified in two groups: non-relational domains that find numerical properties on variables individually, and relational domains that permit to find numerical properties on logically linked variables. Non-relational domains include for example the interval domain [16] (wich permits to find invariants of the form $v_i \in [c_1, c_2]$ where v_i is a variable of the program and c_1 and c_2 are numerical constants), the constant propagation domain ($v_i = c$) and the congruence domain [17] ($v_i \in a\mathbb{Z} + b$). Example of relational domains can be cited such as the polyhedron domain [18] ($\alpha_1 v_1 + ... + \alpha_n v_n \leq c$), the linear equality domain [19] ($\alpha_1 v_1 + ... + \alpha_n v_n = c$) and the linear congruence equality domain [20] ($\alpha_1 v_1 + ... + \alpha_n v_n \equiv a[b]$). The problem with relational domains is that the algorithms they use usually do not scale on large programs. That is why *Frama-C* uses computational methods that are based on non-relational domains.

```
00: extern int a, b;
01: void f(int);

03: void g(){
04:   if (b == 0) a = 1;
05:   else if(b == 1) a = 2;
06:   else return;


09:   f(a);
10: }
```

```
00: extern int a, b;
01: void f(int);

03: void g(){
04:   if (b == 0) a = 1;
05:   else if(b == 1) a = 2;
06:   else return;

08:   assert((a == 1 && b == 0) || (a == 2 && b == 1))
09:   f(a);
10: }
```

Fig. 2. *C* code sample that emphasises relations between variables

SIDAN Plugin in the Frama-C framework We implemented in *SIDAN* the computation of numerical constraints for a given system call. *Frama-C* provides a *Value Analysis* plugin that is able to provide a computation of the variation domains of the variables that influence the function calls. This plugin provides constraints of the type *"integer variable x lies within the domain* [0,5] *in all executions"* as a result. If we consider

the example Figure 2, the assertion generated for the call to the function f, using the *Value Analysis* plugin of *Frama-C* alone would be $a \in \{1,2\}$ and $b \in \{0,1\}$. Indeed, as the Value Analysis plugin uses a non-relational abstract domain, his result misses the relation between the variables a and b.

If we consider the program Figure 2, we see that when $b == 0$ then $a == 1$, and when $b == 1$ then $a == 2$. Actually, to obtain this result we have to consider that there are two paths leading to the call to the function f, and that the constraint to verify at the call to f should take these two paths into account. The *Value Analysis* plug-in uses an algorithm that can potentially keep in memory several invariants computation on several execution paths. The plug-in can be parametrized to define the number of paths explored in parallel by the *Value Analysis* plug-in, which is related to the number of states it keeps in memory before computing an union. If the number of paths explored in parallel is sufficient, the *Value Analysis* plug-in now has internally the information required to build these kinds of constraints. By using a hook in the *Value Analysis* plug-in, it is possible for our plug-in to access this internal information while the analysis is performed. Thus, it allows us to build the invariant by using the variation domain of all the variables on each path. In the example we have described, the invariant generated for line 08 is $(((b == 0) \wedge (a == 1)) \vee ((b == 1) \wedge (a == 2)))$.

Note that the example we give here focuses on invariants computed for integers. In practice, the *Value Analysis* plug-in performs well on integers and floats, but is not very efficient for pointer analysis (at best, it detects access to an unallocated buffer and some out-of-bound access). Discovering constraints on strings is also unavailable due to the fact that the specification of the standard string functions is not included in *Frama-C*. In order to build some constraints on strings, we have preprocessed the source code to replace standard string comparisons by a set of character comparisons whenever possible (see Figure 3 line 3). As a result, some constraints on string buffers have been obtained in the programs we tested our approach on.

3.3 Generated assertions

In order to verify the constraints in the program, we insert executable assertions (see Figure 2 line 08), which is a technique heavily used in the dependability domain, and more precisely in defensive programming [21,22].

The constraints that we can compute for a given system call deal with the variables that are available locally in the context of the system call. However, this call generally depends not only on the local variables but also on the variables manipulated by previous functions in the call stack. That is why it is necessary to compute invariants for all function calls that are on the path that leads to the system call. This implies that we must distribute the executable assertions on all the paths that lead to system calls. Moreover, some system calls can be performed in functions located in external libraries. As a consequence, we choose to insert executable assertions in front of each function call.

To demonstrate the assertion generation capabilities of our data-oriented detection model, we first use as an example a vulnerable version of *OpenSSH*.

The code in Figure 3 is inspired by this vulnerable version of *OpenSSH* and reproduces the basic structure of the real code. The vulnerability is located in the *packet_read* function and can be used to overwrite the value of the *passwd* variable with an empty

```
00: void do_authentication(){
01:   int auth = 0;
      ...
03:   if(!strcmp(pwd, ""))
        /* for users with no password */
05:   else
        /* do_authloop(); */
07:     while(auth != 1) {
08:       type = packet_read(data);
09:       switch (type) {
10:         case SSH_CMSG_AUTH_PASSWORD:
11:
12:           auth = auth_password(pwd, data);
13:           break;
14:        ...
15:       }
16:     }
17:   do_authenticated(user);
18: }
```

```
00: void do_authentication(){
01:   int auth = 0;
      ...
03:   if(pwd[0] != '\0')
        /* for users with no password */
05:   else
        /* do_authloop(); */
07:     while(auth != 1) {
08:       type = packet_read(data);
09:       switch (type) {
10:         case SSH_CMSG_AUTH_PASSWORD:
11:           assert(pwd[0] != '\0');
12:           auth = auth_password(pwd, data);
13:           break;
14:          ...
15:       }
16:     }
17:   do_authenticated(user);
18: }
```

Fig. 3. Example inspired from *OpenSSH*

	OpenSSH	DropbearSSH	ihttpd	fnord	ssmtp
Number of lines	38000	11000	1043	2303	2976
Number of assertions	291	91	145	41	240
Computation time	6 hours	3 hours 45 minutes	1 minute 17 seconds	5 hours	22 minutes

Table 1. Assertions generated

string during the execution of *do_authloop*. This allows a successful authentication on the system with any known account (e.g., *root*) and without having to provide a valid password.

Among the assertions generated, the one located at line 11 in the example in Figure 3 has been produced by our plug-in and detects this attack against the program state.

In order to figure out the capability of our tool to generate assertions on common programs, we have applied it on SSH servers (*OpenSSH* and *Dropbear SSH*), http servers (*fnord* and *ihttpd*), and a smtp server (*ssmtp*). The results are summarized in Table 1. As a result, we could say that the number of assertions generated is obviously heavily dependant on the program source code.

4 Assessment of the detection mechanisms

Even though it is possible to test our detection mechanism against various real world attacks such as those described in [10], such a method would only cover a very small subset of all possible attacks. In order to evaluate the detection coverage of our approach, we would need to know all the vulnerabilities that afflict a program as well as every possible way of exploiting them. As it is not possible to automatically compute this from the source code, we need to define another method to evaluate the detection coverage of our model. In this section we propose a method to assess the detection mechanisms by simulating attacks against non-control-data items without prior knowledge of the vulnerabilities. Our goal is to simulate the consequences of non-control-data attacks by directly modifying in the process memory space the data items it is currently

manipulating. In this section, we propose an approach to evaluate our detection mechanism that is similar to the ones proposed in the security field to help discover new vulnerabilities (fuzzing) and in the dependability field to evaluate fault detection and tolerance mechanisms (fault injection).

4.1 Simulation of attacks against non-control data

Generally speaking, a particular vulnerability usually allows the attacker to access a limited part of a process memory. However, in the worst case scenario it can give to an attacker an access to the whole memory space of a process. For that reason, our injection mechanism is given access to potentially every internal data item of the program under test. However, to accurately simulate a real non-control-data attack, we want to restrict (1) the locations and (2) the instants where an injection can occur during the execution of a program. Firstly, during such an attack not every data item is a potential target. The data items that may be of interest for an attacker are within the subset of data items that can influence the execution of the system calls. Consequently, we target only these data items (they define the locations of potential injections). Other data items are irrelevant for our simulation approach. Secondly, we will modify such items only when they are currently in use (i.e., when they are influencing the current execution of the program).

4.2 Code instrumentation and fault injection

To simulate this injection model, two problems have to be addressed: how do we determine the set of data items that are potential targets for a non-control-data attack, and how do we determine for each one of them when it is appropriate to inject a corrupted value. Clearly, the set of data we want to modify is the very same set of data items we have defined in Section 3.2.

The simplest way to determine the memory address of a variable we want to inject is to obtain it at execution time. This is why we have choosen to also embed the corrupting mechanisms within the source code. Moreover we have decided to distribute the injection mechanisms when the corresponding variables are reachable, that is right before every function call that depends on them. We used the same approach as described in Section 3.3 where we discussed the distribution of the detection mechanisms. In the end, each candidate function call is preceded by a call to the corrupting function implemented by a single external function called *inject()*.

Each injection point is assigned a unique identifier. This identifier is passed as a parameter to the injection function. The remaining arguments are the number of variables that can be corrupted and for each one of them, its address and its size. The corrupting function is controlled by an external process using environment variables. This process controls the unique identifier of the injection that is to be activated, the variable that will be corrupted and the value used to perform the injection. An injection is triggered only once, even when the call to the corrupting function happens many times (*e.g.*, in a loop). The tool presented in Section 3 has been modified in order to perform the instrumentation needed by our injection mechanism. Note that the set of variables A used in an assertion is always a subset of the set of variables I used in the injection process (see Figure 4). Indeed, the injection can be performed in any variable that influences

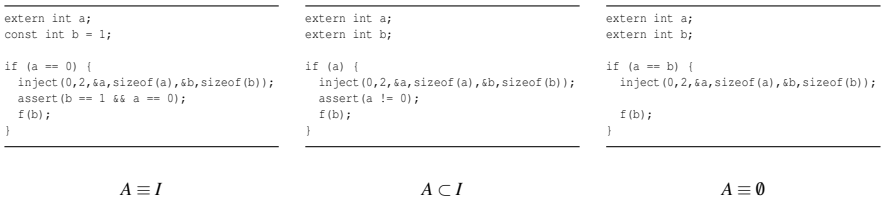

Fig. 4. Different cases of injections and assertions

the function call, unlike the assertions that only concern variables for which value constraints have been discovered.

Our goal is to evaluate our detection mechanism presented in Section 3. To do that, we need to cover a large set of memory corruptions that might be used by a malicious user to perform an intrusion. Very much like a fuzzing technique, we are going to randomly put the internal state of the process in an erroneous state. We perform various injections during the execution of the program used in our test environment in order to simulate the result of a vulnerability exploitation.

To activate a maximum of function calls in the program, we have written a set of scenarios whose goal is to make the control-flow pass through a maximum number of function calls. In the case of *Dropbear SSH*, we have written a set of 24 scenarios that allows us to reach 92% of the function calls.

During the injection process, for each function call that can be reached by a scenario, a random variable from the set of variables that influences the execution of this function call is chosen to be injected with a random value. Each time an attack is simulated, the controller logs if the scenario ended properly or if the process exited unexpectedly or found itself in a deadlock and needed to be killed after a time-out. The controller also logs the behavior of the process during the attack (in terms of system calls and their arguments). The whole test setup is shown in Figure 5.

4.3 Evaluation results

Using the experimentation protocol described in Section 4.2, we have performed a total of 120 000 injections on the *Dropbear SSH* server. As explained before for each injection, we have logged three kinds of information. Firstly, we have compared the output generated by the server during the injection with respect to the output generated without injection. These observations can be considered as an extremely accurate indicator of a potential attack. Indeed, in these cases, the modification of a single variable has been able to modify the execution of the SSH server upto the point its external behaviour (as seen by an SSH client) was changed. Note that 69.36% of the injections have lead to such an alert (either a deviation of output, or a crash of the server). Of course, while being an extremely accurate way of detecting intrusions, this approach is difficult to generalize in real life settings, since it would require to compare the output produced by the server for each command it receives with a reference output. Considering the generally extremely large set of outputs such a server can produce, this

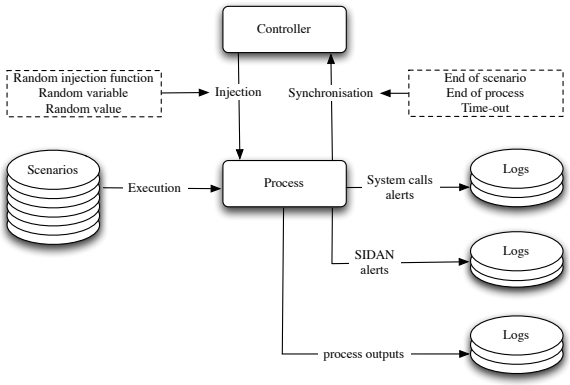

Fig. 5. Experimentation protocol

	SIDAN alert	Unexpected server exit	Incorrect server output	Strace alert
Injection detected	74827	21574	61470	26970
Detection rate	62.36%	18.13%	51.23%	22.48%

Table 2. Injection results on *Dropbear SSH*

approach is hopeless. Here we were able to use such an approach because of the limited set of scenarios we have used during the assessment. Secondly, we also recorded the set of system calls (with their arguments) that were generated during normal executions of the different scenarios (training), and during injections. These recordings have been submitted afterward to an offline intrusion detection mechanism [3]. Once again, this IDS was settled in optimum conditions, since it was trained for a given scenario. And even in these optimal conditions, note that it only detects 22.48% of injections.

Finally we have recorded the alerts generated by our SIDAN tool. The results of all these measures are summarized in Table 2. A more detailed version of the obtained results is given by the Figure 6. We can see that SIDAN detects 62.36 % of the injections. This detection rate is comparable to the one obtained by the first IDS based on the comparison of the output generated by the server (but recall here, that we claim that this kind of IDS is extremely difficult to build in real settings). However, SIDAN is still prone to false negatives with at most 37.64% of injections missed. We can refine these figures by taking into account the fact that within these 37.64% of cases where SIDAN raised no alert, 10.63% where cases where : (1) neither the output generated by the SSH server deviated from the reference output. (2) nor the system call trace deviated from the reference trace. We can be highly confident that these cases do not correspond to exploitable attacks. Hence we can subtract these 10.63% from the figures obtained for false negatives for SIDAN. All in all, we can claim that the rate of false negatives for SIDAN lies within a 27.01% and 37.64%.

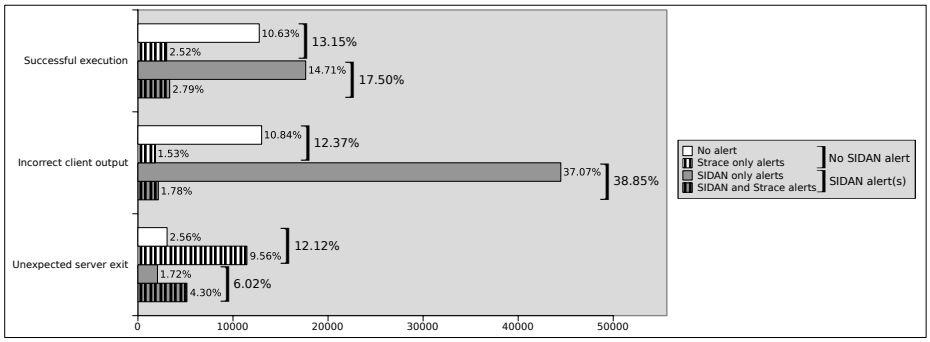

Fig. 6. Distribution of alerts

5 Conclusion and future work

In this article, we propose a sofware-level intrusion detection approach based on the internal state of the process that detects data attacks, which are missed by traditional control-flow approaches. Our mechanism relies on a data-oriented behavior model to detect erroneous states that could lead to illegal system calls. We present a tool that implements our approach by analyzing and instrumenting a program's source code. This tool has proved that our approach is useable in the context of real software and that it can detect real world non-control-data attacks (such as the null password attack on *OpenSSH*). We also propose a method to assess these intrusion detection systems against data attacks by using a fault injection mechanism. In the particular case of *Dropbear SSH*, by using our evaluation method, we have estimated, without prior knowledge of any attacks, an approximation of the detection coverage of our detection model.

However, the current implementation of our tool computes the constraints needed by our detection model using only variation domains. This is clearly a limitation, because it does not permit the detection of data attacks on variables whose variation domain is statically unknown in the source code. That is why in the future we intend to use additional static analysis techniques to discover more constraints. We also plan to investigate for our evaluation method the possibility of replacing the set of hand written scenarios by automatically generated scenarios using fuzzing techniques [23].

References

1. Hofmeyr, S.A., Forrest, S., Somayaji, A.: Intrusion detection using sequences of system calls. Journal of Computer Security (1998)
2. Kruegel, C., Kirda, E., Mutz, D., Robertson, W.: Automating mimicry attacks using static binary analysis. In: 14th conference on USENIX Security Symposium. (2005)
3. Kruegel, C., Mutz, D., Valeur, F., Vigna, G.: On the detection of anomalous system call arguments. In: 8th European Symposium on Research in Computer Security. (2003)
4. Bhatkar, S., Chaturvedi, A., Sekar, R.: Dataflow anomaly detection. In: 2006 IEEE Symposium on Security and Privacy (S&P'06). (2006)

5. Mutz, D., Robertson, W., Vigna, G., Kemmerer, R.: Exploiting execution context for the detection of anomalous system calls. In: Proceeding of the 10th International Symposium on Recent Advances in Intrusion Detection. (2007)
6. Feng, H., Kolesnikov, O., Fogla, P., Lee, W., Gong, W.: Anomaly detection using call stack information. In: 2003 IEEE Symposium on Security and Privacy. (2003) 65
7. CEA: Frama-c, framework for modular analysis of c
8. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity. In: CCS '05: Proceedings of the 12th ACM conference on Computer and communications security. (2005)
9. Kiriansky, V., Bruening, D., Amarasinghe, S.: Secure execution via program shepherding. In: Proceedings of the Usenix Security Symposium. (2002)
10. Chen, S., Xu, J., Sezer, E., Gauriar, P., Iyer, R.: Non-control-data attacks are realistic threats. In: Usenix Security Symposium. (2005)
11. Akritidis, P., Cadar, C., Raiciu, C., Costa, M., Castro, M.: Preventing memory error exploits with *wit*. In: 2008 IEEE Symposium on Security and Privacy. (2008)
12. Castro, M., Costa, M., Harris, T.: Securing software by enforcing data-flow integrity. In: 7th USENIX Symposium on Operating Systems Design and Implementation. (2006)
13. Demay, J.C., Totel, E., Tronel, F.: Sidan: a tool dedicated to software instrumentation for detecting attacks on non-control-data. In: 4th International Conference on Risks and Security of Internet and Systems (CRISIS'2009), Toulouse (October 2009)
14. Weiser, M.: Program slicing. IEEE Transactions on Software Engineering (1982)
15. Kuck, D.J., Kuhn, R.H., Padua, D.A., Leasure, B., Wolfe, M.: Dependence graphs and its use in optimization. In: 8th ACM Symposium on Principles of Programming Languages. (1981)
16. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. (1977)
17. Granger, P.: Static analysis of arithmetical congruences. International Journal of Computer Mathematics **30** (1989) 165–190
18. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. (1978)
19. Karr, M.: Affine relationships among variables of a program. In: Acta Informatica. (1976) 133–151
20. Granger, P.: Static analysis of linear congruence equalities among variables of a program. In: TAPSOFT'91. (1991) 169–192
21. Goloubeva, O., Rebaudengo, M., Reorda, M.S., Violante, M.: Soft-error detection using control flow assertions. In: Proceedings of the 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'03). (2003)
22. Vemu, R., Abraham, J.A.: Ceda: Control-flow error detection through assertions. In: Proceedings of the 12th IEEE International On-Line Testing Symposium). (2006)
23. Neves, N., Antunes, J., Correia, M., Verissimo, P., Neves, R.: Using attack injection to discover new vulnerabilities. Conference on Dependable Systems and Networks (2006)