Building an application data behavior model for intrusion detection

Olivier Sarrouy and Eric Totel and Bernard Jouga

Supelec, Avenue de la Boulaie, CS 4760, F-35576 Cesson-Sévigné CEDEX, France
E-mail : firstname.lastname@supelec.fr

Abstract Application level intrusion detection systems usually rely on the immunological approach. In this approach, the application behavior is compared at runtime with a previously learned application profile of the sequence of system calls it is allowed to emit. Unfortunately, this approach cannot detect anything but control flow violation and thus remains helpless in detecting the attacks that aim pure application data. In this paper, we propose an approach that would enhance the detection of such attacks. Our proposal relies on a data oriented behavioral model that builds the application profile out of dynamically extracted invariant constraints on the application data items.

1 Introduction

Two approaches coexist in intrusion detection. The misuse-based approach relies on the research of known attack signatures in the data collected over the information system. Because it requires signatures, this approach cannot detect unknown attacks. On the other hand, the anomaly-based approach relies on the comparison, at run time, of the application behavior with a previously built normal behavior model. At the application level, the anomaly-based approach is largely preferred. In this case, the normal behavioral model of an application is often built dynamically by the observation of the sequences of system calls it emits [11]. Such system call based intrusion detection systems appear to be insufficient as they cannot detect anything but control flow violations. Thus, attacks that do not disturb the control flow of the application and focus on pure data remain undetected [6].

In this paper we introduce an approach that would enhance the detection of such attacks. Our proposal relies on a data oriented behavioral model. More precisely, our goal is to dynamically discover invariant constraints on the application data that would characterize normal states of the application. This paper is organized as follows : we first discuss the existing work on the topic, we then explain how to build the application data model, and we finally exhibit our prototype implementation before discussing the results we obtain.

2 Related work

Most recent application level anomaly-based intrusion detection systems rely on the immunological approach introduced by Forrest and al. [11]. This approach

is built over an external - or *black-box* - behavioral model and consists in monitoring the sequences of system calls emitted by the application [13]. Such intrusion detection systems are efficient in detecting classical attacks which obviously modify the sequences of system calls emitted by the application, but are easy to evade by mimicing the sequence of system calls the application is supposed to emit [20]. To contend this kind of attacks, proposals have been done to enhance the behavioral model with process internal information, such as the content of the call stack or the value of the program counter at the time of system calls [12]. Such approaches, called *gray-box approaches*, indeed make mimicry attacks more difficult to succeed but remain unable to detect anything but control flow integrity violation. However, another kind of attacks exists which does not disturb the control flow of the application but nonetheless leads to the same threat than classical attacks [18,6].

This second class of attacks focuses on the data that do not influence the application control flow, and are thus called *pure data attacks* or *non-control data attacks*. These attacks constitute an important threat as they cannot be detected by any of the current intrusion detection systems [18]. Furthermore, it appears that numerous real world vulnerabilities can be exploited using such a pure data attack [6]. Some work has already been done which aims at detecting pure data attacks, mainly focusing on the integrity of the application data-flow, either through complete data-flow graph or through taint-checking [4,5,17,14]. In this paper, we propose to dynamically discover likely invariants in the application data in order to characterize its normal behavior. In this goal, we rely on the definition of the set of data that may be sensitive to an intrusion attempt and thus on the notion of tainted data.

3 Pure data attacks

As explained in the previous section, current intrusion detection systems remain helpless in detecting what we call pure data attacks, *i.e.*, attacks which do not violate the integrity of the application control flow. In this section we first introduce an example of such an attack and then study it in the perspective of the properties it breaks concerning the data it modifies.

3.1 An example of a pure data attack

A typical example of a pure data attack may be found in the exploitation of the WU-FTPD *Site Exec Command Format String* Vulnerability [3] described in [6]. This vulnerability allows an attacker to overwrite a `C` structure, denoted `pw`. The severity of this vulnerability resides in that the `pw` structure contains the `uid` of the authenticated user. More precisely, this structure is used to re-affect the application rights to those of the authenticated user after each operation which requires the application to gain root privileges, such as the `setsockopt()` system call used in the treatment of a `GET` request. This example of complete attack, given in Figure 1, is interesting as it clearly does not disturb the control

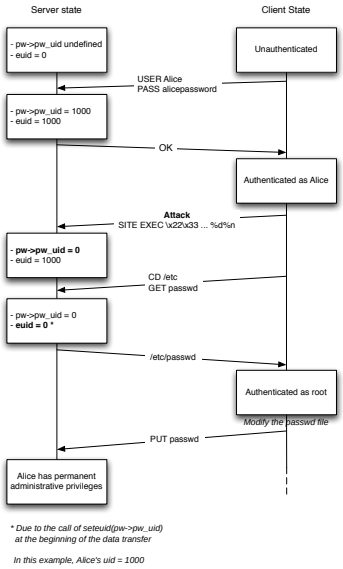

Fig. 1. WU-FTPD pure-data attack

flow of the application, and thus would not be detected by any of the classical system call based approaches. An other interesting point in the study of this attack is that it appears to be characteristic of the properties pure data attacks break when they are performed. We discuss this idea in more details in the next subsection.

3.2 Attack characterization

The pure data attack described in the previous subsection appears to be interesting in how characteristic it is in the disturbance of the expected properties of the application data. Indeed, this attack breaks a very simple property verified during a normal use of the application, which is that the `pw->pw_uid` variable should remain constant during a given session. In this perspective, denoting $\texttt{pw->pw\_uid}_1$, $\texttt{pw->pw\_uid}_2$, ..., $\texttt{pw->pw\_uid}_n$ different values of `pw->pw_uid` during the execution of a given session, this attack may be characterized by the fact that whenever $\texttt{pw->pw\_uid}_1$, $\texttt{pw->pw\_uid}_2$, ..., $\texttt{pw->pw\_uid}_n$ are extracted, it breaks the simple constraint $\texttt{pw->pw\_uid}_1 = \texttt{pw->pw\_uid}_2 = ... = \texttt{pw->pw\_uid}_n$.

More generally, it appears that most pure data attacks - and even more classical attacks - can be characterized by the fact that they break one of the constraints on data that are verified when the application remains in a « normal » state, *i.e.*, when it runs without being attacked.

4 Data-based intrusion detection model

As explained above, most pure data attacks - as well as more classical attacks - can be characterized by the fact that they break some properties verified by the application data when this application remains in a « normal » state. Thus, we believe that extracting and afterwards controlling these properties would allow us to detect intrusions more accurately. In this section, we first introduce a formal definition of a process state and propose an abstraction of this definition focused on the properties verified by the application data. Then, we intend to define the notion of attack in the eyes of this abstraction, still focusing on the properties verified by the application data. Finally, we propose a detection model based on that previously introduced considerations.

4.1 State of a process

For each discrete time i, the *snapshot state* s_i of a process may be defined by $s_i = \langle pc_i, v_{1_i}, ..., v_{n_i} \rangle$ where pc_i is the location in the process of the executed instruction at time i, i.e. the value of the program counter, and v_{1_i}, ..., v_{n_i} the values of the various data manipulated by the program at time i (registers, environment variables, global and local variables, etc ...). This definition of the state of a process may be insufficient in the perspective of intrusion detection. Indeed, controlling the consistency of snapshot state s_i at a given time i sometimes requires the knowledge of all the previous snapshot states $s_0, ..., s_{i-1}$. We thus define the *global state* S_i of a process - in its temporal meaning - at time i by $S_i = \langle s_0, ..., s_i \rangle = \langle pc_0, ..., pc_i, v_{1_0}, ..., v_{n_0}, ..., v_{1_i}, ..., v_{n_i} \rangle$.

The set of all potential global states of a process, denoted $\mathcal{S}$ is a huge set where all elements cannot be reached during a « normal » execution of a given program. Though, we may define the set of allowed global state of a process, denoted $\mathcal{A}$ as the set of global states which can be reached in a context of an attack free execution.

In practice, it appears that $\mathcal{A}$ constitutes a quite small subset of $\mathcal{S}$ that we would like to define in order to discern the « allowed » states and the « unallowed » states. As it seems impossible and probably not much relevant to explicitly define $\mathcal{A}$, we propose to abstract the state definition to implicitly define it by expressing the various constraints on the values that application data can effectively take when this application is in an allowed global state. In other words, we define a number of relationships which constitute a constraint system $\mathcal{C}$. Given a global state $S_i = \langle s_0, ..., s_i \rangle = \langle pc_0, ..., pc_i, v_{1_0}, ..., v_{n_0}, ..., v_{1_i}, ..., v_{n_i} \rangle$ we consider that $S_i \in \mathcal{A} \iff \langle pc_0, ..., pc_i, v_{1_0}, ..., v_{n_0}, ..., v_{1_i}, ..., v_{n_i} \rangle$ verifies $\mathcal{C}$. Given this definition a « normal » application state, we can now propose an attack model.

4.2 Attack model

As explained above, we may assume that the application is in an unallowed global state, *i.e.*, a state $S \in \mathcal{S} \backslash \mathcal{A}$ when one of the constraints of $\mathcal{C}$ is broken.

Thus, we propose, on the basis of our data constraint based state model, a definition of an attack as a sequence of « user actions » which leads the application from a state $S_i \in \mathcal{A}$ into a state $S_f \in \mathcal{S} \backslash \mathcal{A}$, therefore breaking the constraints on the application data that characterize the set $\mathcal{A}$. Given this definition of an attack, we can now propose a detection model which focuses on the constraints verified by the application data.

4.3 Detection model

Our detection model relies on the assumption that it is possible to extract in whatever manner the set of constraints $\mathcal{C}$ which characterize the set of allowed global states $\mathcal{A}$. Supposing this assumption verified, we thus propose to monitor the application by controlling the enforcement of this set of constraints $\mathcal{C}$ and to raise an alert as soon as one of these constraints seems broken. However, the extraction of the set of constraints $\mathcal{C}$ may appear quite complex. Nevertheless, as our goal is not to fully qualify the consistency of a given state, but only to qualify it in a security perspective, it is clear that not all the data items manipulated by the application are interesting for our work. In the next section, we thus try to define which data can be critical in a security focused perspective and then examine a way to practically extract this data out of the whole application data set.

5 Intrusion sensitive data set

The set of interesting data we try to extract, henceforward called *intrusion sensitive data set* and noted *ISDS*, is defined by the two main properties it verifies. First, in an immunological approach, it constitutes a subset of the data which may influence the system calls. Furthermore, it constitutes a subset of the data being influenced by user inputs, which we call *tainted data*. The notion of influence between two or more data can be formally defined by the notion of causal dependency [8,19,7]. Denoting *(o,t)* the content of the data object o (a byte or a variable depending on the level of granularity) at time t, we may then denote *(o',t')* $\rightarrow$ *(o,t)*, with $t' \leq t$ the causal dependency of *(o,t)* in relation to *(o',t')*. The relation $\rightarrow$ being transitive, we may then define the causality cone of a point *(o,t)* as $cause(o,t) = \{(o',t')/(o',t') \rightarrow (o,t)\}$. In the same way, we may define the dependency cone *dep(o,t)* as the set of points which causally depend on *(o,t)* and write $dep(o,t) = \{(o',t')/(o,t) \rightarrow (o',t')\}$.

This notion being introduced we may now give a more formal definition of the intrusion sensitive data set. The first property characterizing the intrusion sensitive data set expresses the fact that *ISDS* belongs to the causality cone of the system calls and their arguments. The second property characterizing the intrusion sensitive data set as well expresses the fact that *ISDS* belongs to the dependency cone of user inputs. Thus, as we have expressed *ISDS* as the set of data respecting these two properties, we may define it as the intersection of both the causality cone of the system calls or their arguments and the dependency

cone of the user inputs and denote (with *sc* the set of system calls and *ui* the set of user inputs) :

$$ISDS = cause(sc) \cap dep(ui)$$

In the next section, we present the kind of constraints we aim at extracting out of the intrusion sensitive data set we have just defined.

6 Constraints determination

In section 4.2, we have formulated the hypothesis that attacks generally violate invariant constraints on the application data. We have thus tried to extract these constraints, with the help of an automatic invariant discovery tool called *Daikon* [1,10,9]. Daikon analyzes the execution traces of a given application and tries to extract invariant properties out of it on the basis of a property grammar which contains the set of all searched invariants. These invariants, which are thus exhaustively verified on the execution trace data, can be very complex : constant data item (e.g. x = a), data item taking only a few distinct different values (e.g. x ∈ {a, b, c}), definition set (e.g. x ∈ [a..b]), non-nullity, linear relationship (e.g. x = ay + bz + c), order relationship (e.g. x ≥ y), single invariants on x + y, x - y, etc. We introduce in the next section the implementation of our prototype, and explain how to generate the execution traces needed for a Daikon analysis.

7 Implementation

In order to generate the execution traces on *ISDS*, we have used Valgrind [2,16] to emulate a processor controlled by our prototype and have executed the monitored application on this emulated processor. Valgrind is a dynamic binary instrumentation framework offering a dedicated API to emulate a processor and study the binaries executed on this emulated processor. Our intrusion detection system prototype, called *Fatgrind*, has thus been designed as a plugin to Valgrind. Fatgrind mainly aims at dynamically extracting *ISDS* and generating its execution traces containing the value of the data belonging to it. To achieve these goals, Fatgrind builds a shadow of the memory of the monitored process [15]. Each time a byte of memory is written, Valgrind checks whether it is tainted and thus whether it must be shadowed or not. Furthermore, when a system call is emitted, Fatgrind computes its causality cone. Each tainted byte belonging to this causality cone belongs to *ISDS* and is thus dumped into a file. Once enough executions of a given application have been done, the execution traces generated are considered complete enough and are analyzed by *Daikon* to automatically extract the likely invariants it contains. Of course, the quality of the extracted invariants depends on the exhaustivity of the normal application behavior learning.

8 Results

To evaluate our model, we have studied a little snippet of code equivalent to the one studied in section 3. This snippet of code was designed to be representative enough although generating very few traces and thus very few invariant properties, allowing us to easily check their consistency. We have thus focused here on the validation of our approach by trying to control one by one the invariants inferred by Daikon out of the execution traces generated by Fatgrind. Among the various properties extracted we indeed discovered the constraints expressing the equality of the `uid` variable at the beginning of the daemon loop and at its end. The obtained results have thus shown the viability of the approach on a small but representative example. Moreover it appears that most of the attacks described in [6] would be detected by our approach excepted the attack against the SSH server where the attacked data item is whatever modified during a normal use of the application. This report thus encourages us to pursue the prospective work we have engaged even if a lot of enhancements could be brought to our approach.

9 Conclusion and future work

The work presented in this paper proposes a way to enhance application level intrusion detection by introducing a data-oriented detection model. This approach relies on the automatic generation of a behavioral model based on the relationship between application data aiming at detecting state inconsistency at runtime. In order to pursue such an approach, it is necessary to determine which data are sensitive to intrusions and how these data items are related to each other. As shown by the results of the previous section, the proposed approach indeed enables the detection of pure data attacks. However, regarding the conclusion of this prospective work, it appears that several enhancements could be brought. Indeed, the binary level does not allow us to access a high semantical level, as, for instance, we do not get any information about the type of the manipulated data. We therefore plan to apply this approach to programming language using a native intermediate representation (Java, .Net, PHP, etc.) in order to directly modify the interpreter and thus access a richer semantic.

10 Acknowledgement

This work has been funded by the french DGA (General Delegation for Armament) and the french CNRS (National Center for Scientific Research) in the context of the DALI (Design and Assessment of application Level Intrusion detection system) project.

Références

1. Daikon. groups.csail.mit.edu/pag/daikon/.

2. Valgrind. www.valgrind.org.
3. Cert advisory ca-2001-33 multiple vulnerabilities in wu-ftpd. http ://www.cert.org/advisories/CA-2001-33.html, 2001.
4. M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, 2006.
5. L. Cavallaro and R. Sekar. Anomalous taint detection. Technical report, Secure Systems Laboratory, Stony Brook University, 2008.
6. S. Chen, J. Xu, E. Sezer, P. Gauriar, and R. Iyer. Non-control-data attacks are realistic threats. In *Usenix Security Symposium*, 2005.
7. B. d'Ausbourg. Implementing secure dependencies over a network by designing a distributed security subsystem. In *Proceedings of the European Sysmposium on Research in Computer Security (ESORICS'94)*, 1994.
8. D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 1976.
9. M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 2001.
10. M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2007.
11. S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A Sense of Self for Unix Processes. In *Proceedings of the 1996 IEEE Symposium on Research in Security and Privacy*, 1996.
12. D. Gao, M. K. Reiter, and D. Song. Gray-box extraction of execution graphs for anomaly detection. In *Proceedings of the 11th ACM conference on Computer and communications security*, 2004.
13. S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 1998.
14. E. Larson and T. Austin. High coverage detection of input-related security faults. In *Proceedings of the 2003 Usenix Conference (Usenix'03)*, 2003.
15. N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. *Proceedings of the Third International ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, 2007.
16. N. Nethercote and J. Seward. Valgrind : A framework for heavyweight dynamic binary instrumentation. In *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, 2007.
17. J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS 2005)*, 2005.
18. C. Parampalli, R. Sekar, and R. Johnson. A practical mimicry attack against powerful system-call monitors. Technical report, Secure Systems Laboratory, Stony Brook University, 2007.
19. A. Sabelfeld and A. Myers. Language-based information-flow security, 2003.
20. D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *CCS '02 : Proceedings of the 9th ACM conference on Computer and communications security*, 2002.