Automated Instruction-Set Randomization for Web Applications in Diversified Redundant Systems

Frédéric Majorczyk
IRISA/University of Rennes 1
Rennes, France
frederic.majorczyk@irisa.fr

Jonathan-Christofer Demay
SUPELEC
Rennes, France
jonathan-christofer.demay@supelec.fr

Abstract
The use of diversity and redundancy in the security domain is an interesting approach to prevent or detect intrusions. Many researchers have proposed architectures based on those concepts where diversity is either natural or artificial. These architectures are based on the architecture of N-version programming and were often instantiated for web servers without taking into account the web application(s) running on those. In this article, we present a solution to protect the web applications running on this kind of architectures in order to detect and tolerate code injection intrusions. Our solution consists in creating diversity in the web application scripts by randomizing the language understood by the interpreter so that an injected code can not be executed by all the servers. We also present the issues related to the automatization of our solution and present some solutions to tackle these issues.

1 Introduction

Since the seminal paper of Forrest [13], diversity is considered as a promising approach to prevent intrusions such as the spreading of worms. Some researchers have proposed different techniques to automatically create diversity by randomizing instruction set [2,16], randomizing address space [4,29] or randomizing system calls [8]. Although these techniques are efficient against existing attacks, there are ways to circumvent these protections [24,25] by guessing the secret on which these techniques rely.

Redundancy is a classical approach to provide safety in different systems. It is efficient to tolerate physical faults. On the other hand, redundancy without diversity is inefficient against design faults which can be exploited by an attacker to compromise the whole system. In a redundant system, it is necessary to diversify at least some components to detect or tolerate intrusions. Two articles advocate the use of redundancy and diversity in the security field [11,17].

Some projects explored this approach by using either natural diversity or artificial diversity. All the architectures proposed are based on the architecture of N-version programming [1], a design diversity [12] technique. In N-version programming, different variants called versions are designed and developed independently from the same specification so that the probability of common faults is as low as possible. These versions are executed in parallel; their results are compared and a vote allows to determine the correct result. The versions should indeed behave the same way and produce the same outputs except when a fault is activated in one of them. Developing versions specifically is expensive: researchers have proposed either the use of COTS (Components-Off-The-Shelf) to replace the versions or the use of techniques to create diversity from one COTS.

On the one hand, some projects [7,14,15,26–28] have proposed the use of natural diversity, that is, COTS instead of specifically developed versions. They compare the outputs of the COTS [26] or the sequences of system calls [14]. Depending on the compared elements, the algorithm is more or less complicated. This kind of architecture probabilistically protects against binary code injection, logic faults in the source code and configuration faults.

On the other hand, researchers have proposed to use different techniques to diversify automatically a single COTS [3,9,23]. Once diversified, they compare the operating systems IO buffers [3] or the system calls and their arguments [9,23]. These projects protect against binary code injections (stack overflows and heap overflows).

Many of these projects implement this for web servers but do not take into account the web applications running on those servers, while a large number of vulnerabilities affect web applications and not web servers. Roberston et al. [22] estimate that the percentage of vulnerabilities in CVE related to web applications between 1999 and 2005 is about 25%. In all these projects, the problem of the web application is not considered, that is, either there is no web application running on the web server or the web application is
not diversified. Without any other mechanisms, it is necessary to diversify the web applications to detect or tolerate intrusions that can affect the servers. Three solutions can be proposed: using N-version programming (developing N times the web applications), using different COTS web applications and artificially diversifying the web applications.

The first solution is certainly the best considering the coverage of the detected and tolerated intrusions but it is expensive and the specification must be precise enough otherwise the comparison algorithm will be too complicated. Using COTS web applications can not be considered because their specifications are too different: for example, there are many bulletin boards available and developed in several different languages but their specifications and features are too different. The only viable solution is to diversify artificially the web applications running on the web servers. That is the solution we explore in this article. We propose the use of a technique described in [6, 16] in a redundant diversified architecture. This solution allows to detect or tolerate high-level code injection intrusions by randomizing the language understood by the interpreter. It can be used on top of the previous work in redundant diversified architectures so as to obtain a comprehensive architecture able to detect or tolerate many classes of intrusions. We study the issues related to the automatization of this technique and propose some solutions to bypass these difficulties.

In next Section, we present the architecture we propose and its properties compared to the architectures previously proposed. In Section 3, we present the issues related to the automated randomization process of the language. We present some solutions that allows to resolve these issues in realistic cases. Section 4 discusses some limitations of our work. Section 5 presents related work.

2 Architecture and Security Properties

The architecture we present here is not new: it is clearly based on the classical architecture of N-version programming. This architecture has been used, with some changes, in many projects [15, 26, 28]. First, we present the architecture and how it works; then we present its properties.

2.1 Architecture

The Figure 1 illustrates the architecture we propose. The main difference with architectures previously proposed is the use of a diversified web application. This architecture is composed of two main entities: a proxy/IDS and a set of COTS servers. The role of the proxy is to handle the client requests. It forwards the request from the client to the COTS servers and then forwards a response to the client following the decision of the IDS. It is the sole part of the architecture accessible directly by the clients. The IDS is in charge of comparing the behaviors of the COTS servers: it can compare their responses [15, 26, 28], system calls’ sequences [14] or the information flow graphs [18]. Furthermore, in our work, it must compare the result status of the interpreter of the script language in which the web application is written. If some differences are detected, it raises an alert and it informs the proxy of the response that has been elected by a voting algorithm.

The COTS servers constitute the core of the architecture: they provide the service to the client. They can be diverse in terms of HTTP service, operating systems and hardware. The web application is diversified artificially by using instruction set randomization [16]. On each COTS server, the language interpreter understands a different language which is derived from the original language of the web application. We explain this transformation in Section 3.

This kind of architectures are very interesting because intrusions can be tolerated if the number of COTS servers is high enough: $2f + 1$ are necessary to tolerate $f$ simultaneous intrusions. The detection and tolerance is possible only if there is no common vulnerabilities affecting the COTS servers. If an intrusion is detected, the server affected by the intrusion must be removed from the architecture and reconfigured to be included again later.

2.2 Security Properties

The detection and tolerance rely on a voting algorithm on the result status of the interpreters running on each COTS server. If a vulnerability allowing a code injection is present in the web application, an attacker can exploit this vulnerability on the N COTS servers and ensure that the IDS will not detect any differences by comparing the chosen elements of comparison (responses of the COTS servers, system calls, etc.). By diversifying the language understood by each interpreter, an attacker will not be able to inject code that will execute on the N COTS servers. At the most, an attacker will be able to inject code in one of the web applications. Our IDS must compare the result status of the
interpreters. Three cases are possible:

- All the interpreters have interpreted successfully the script: there has not been any code injection.
- All the interpreters have failed to interpret the script: there has been an attempt to inject some code in the application. That does not mean no injected code has been executed: an interpreter could have executed some injected code and then failed. An alert must be raised and the attack must be analyzed to determine whether a COTS server has executed some injected code and which one if this is the case.
- One of the interpreter has interpreted successfully the script while the others have failed. There has been an effective intrusion on the server whose interpreter has succeeded. An alert must be raised, this server must be reconfigured, the others must be treated as previously.

The detection of code injections is not probabilistic. If the COTS servers are not diverse, that is, they use the same HTTP service, the same operating system, the same hardware, our technique allows to detect and tolerate only high-level injection code intrusions. Built on top of architectures previously proposed where the COTS servers are diversified, this technique extends the coverage of these architectures. However, intrusions that does not inject code in the application cannot, in general, be detected.

In next Section, we detail the randomization technique that we use; we present the issues related to the automation of this technique and some solutions.

3 Automated Randomization

The process consists in randomizing the grammar of languages used in a script and then accordingly modifying interpreters so that the randomized grammars can be recognized. Any code which has not been randomized, like injected code, will fail to be executed by such interpreters with a high probability. However, a script may contain many interpreted languages, such a SQL requests or shell commands embedded within a PHP or a PERL code, and this might pose problems.

3.1 Root Language’s Randomization

We call the language in which others languages can be embedded the root language. Having the proper lexical analyzer, randomizing the root language can be trivial as shown in Figure 2. The script is parsed and every instruction from the root language, including keywords and operators, is concatenated with a random tag. In order to make the root language interpreter recognize this new instruction set, we modify the same way the grammar of its lexical analyzer.

```plaintext
# Generated Perl code
foreach $k ( sort keys %$tre ) {  
v = $tre->{$k};  
die "duplicate key $v\n"  
if defined $list($k);  
push @list, @! $list($k); }
foreach $k ( _OxF sort_OxF keys %$tre )_OxF {  
v =_OxF $tre->{$k};  
die_OxF "duplicate key $v\n"  
if_OxF defined_OxF $list($k);  
push_OxF @list, @! $list($k); }

Figure 2: Randomization process of a PERL sample code
```

3.2 Generated Languages’ Randomization

We call languages that can be embedded in the root language generated languages. It can include the root language itself (the eval function in PHP for example). Randomizing the generated languages is a greater difficult task. They are indeed dynamically generated during execution and so the instructions that need to be randomized may be distributed all along the script within strings from the root language. Therefore, unlike the root language where we can use a lexical analyzer to spot the instructions that need to be randomized, we need to analyze each string in order to determine if it contains parts of generated code and from which language before being able to apply the randomization. Figure 3 is a sample of PHP code which generates SQL code and that might pose problems to the randomization process. Those problems are of two types: a string might look like it contains instructions from a language but it does not (line 2), and it also can be used both as a code-generating string and as standard string (line 3 and 4).

The simplest solution to address those problems would be having web applications to comply with simple programming guidelines. Those guidelines would ensure that developers would only use instructions from generated languages in strings that will actually be used to generate code. Therefore, an automated randomization process could be achieved by using regular expressions as a heuristics to detect those strings and the generated language they contain before applying a randomization algorithm similar to the one used for the root language. The code sample in Figure 3 is a tricky one and this approach might work on lot of available web applications.

However, when this kind of tricky code is present and cannot be easily avoided, we need another solution. A simple proposal would be to fall back to a semi-automated process. The developers would have to manually tag the strings so that the randomizing algorithm would know which ones need to be processed. This solution works to distinguish between ambiguous strings, which are used as standard strings or as code-generating strings, but it cannot handle the case
Figure 3: PHP/SQL sample code

Figure 4: Illustration of the causal dependency cones method

Figure 5: Randomization of an embedded SQL sample code

instructions from a generated language will not appear in any of these cones. The set of strings in all these causality cones, denoted $S_0$, contains the actual strings that need to be instruction-randomized. Like in the simple automated process, we use a heuristic based on regular expressions to determine the language in each string. However, before applying the randomizing algorithm to these strings, we need to detect those which might be used both as a standard string and as a code-generating string.

For each string $s_0 \in S_0$, we compute $\text{dep}(s_0, t_{n_0})$. The set of strings in all these dependency cones, denoted $S_1$, contains the strings that will be affected, rightly or not, by the randomization of $S_0$. Therefore, the strings that would be wrongly affected by the randomization of $S_0$ are in the set $S_1 \setminus S_0$, denoted $S_2$. To avoid this, we need to duplicate every string in the partial information flow common to $S_0$ and $S_2$. To determine which strings need to be duplicated, for each string $s_2 \in S_2$ we compute $\text{cone}(s_2, t_{n_2})$. The set of strings in all these causality cones is denoted $S_3$.

Now the randomizing process can begin. Every strings in $S_0 \setminus S_3$ are directly randomized. Every string in $S_0 \cap S_3$ is duplicated and those duplicates are randomized. Every string from $S_0 \cap S_3$ used in $S_0 \setminus S_3$ is replaced by its duplicate. Figure 4 illustrates this method in the particular example in Figure 3. Figure 5 shows the results of this process.

4 Limitations

Our approach has several limitations. It does not apply to web applications written in compiled languages such as java or C#. This technique does not protect against logic errors in the web application: for example, a directory traversal attack might work on all the COTS servers.

Moreover, it is not possible to use this approach for web applications in which the client sends code to the server. For example, a web application that allows the user to send SQL requests (for an educative purpose for example) will pose problems because the SQL requests cannot be randomized. However, we think that a majority of web applications does not need to receive code from clients.
5 Related Work

Naturally diversified redundant systems. These systems rely on a redundant computation technique that uses the natural availability of multiple implementations of the same standard protocols for many platforms. For example, numerous implementations of the HTTP protocol are available for several platforms. Thus, this approach has been mainly explored on this protocol. For example, in order to detect differences that might indicate an attack, the HACQIT project [15, 21] proposes to compare the HTTP code status between multiple COTS web servers. This idea has then been extended [26–28] to the actual web page responses from the servers. Other ideas have been proposed which consists, instead of focusing on the outputs of COTS, in monitoring their behavior by comparing the sequences of system calls [14] or the information flow graphs [18].

By diversifying the architecture on top of which the web applications are running, this approach provides an efficient defense mechanism against binary code injection. However, since the web applications cannot be diversified the same way without suffering the cost of N-version programming, it is hardly usable against interpreted code injection attacks.

Artificially diversified redundant systems. As it is not always possible to rely on the architecture design to create diversity, numerous techniques have been proposed to automatically produce multiple artificial variations of a program. There is several possible sources of variation in program. For example, the stack can be diversified [23]. Generating one variant that grows the stack in the standard direction, one in the reverse direction, allows the detection stack-based overflow attacks. The source of variation can also be the heap [3]. For example, DieHard is a runtime system that replaces the standard memory manager by one that randomizes the location of objects in the heap. Initializing each variant of a program with a different random seed allows the detection of heap-based overflow attacks. Other sources of variation have been proposed, like address space partitioning [9] which use disjoint memory spaces such that any absolute memory address in one variant is invalid in the other one.

Being automated, this approach provides a lesser degree of diversity than diversified architectures, but is also less likely to cause trouble in the comparison process. However, those techniques cannot generate artificial diversity beside the binary level of a program. Therefore, like the natural diversity approach, it cannot be used as a defense mechanism against interpreted code injection attacks. That’s why in our model, we are combining a binary diversified redundancy approach, with a diversification technique that can be applied to the interpreted level. This diversification technique is based on instruction set randomization.

Instruction Set Randomization. It is a diversity technique aiming at safeguarding against any code injection attack. By generating a program specific randomized set of instructions, it forces any injected code to fail with a high probability. Indeed the attacker have no way to predict the key used to generate this particular set of instructions. This approach can be used as a defense mechanism against binary code injection as well as interpreted code injection [16]. However, the former has be shown to be circumventable in real situation [25] and the later has not been fully explored [16], less likely in a fully automated process [6].

Contrary to those works, first, we study the problem of randomizing the generated languages as well as the root one. We propose several solutions to address the problems issued by the automatization of the process and we propose a theory for a fully automated approach based on causal dependencies [10, 19]. Secondly, by integrating this approach in a diversified redundant system, we nullify the probability of a successful interpreted code injection, even if the attacker is able to break the keys used to generate the randomized sets of instructions.

6 Conclusion

In this article, we present a comprehensive architecture to detect and tolerate both low-level intrusions and high-level code injections. Our architecture is built on top of previous works from the domain of diversified redundant systems for web servers.

We extend the architectures previously proposed to take into account the web applications running on the web servers. It is indeed necessary to diversify the web applications to detect and tolerate intrusions that could affect the servers. To diversify these applications, we propose to use instruction-set randomization and study the issues related to the automatization of the randomization process. We present some solutions to these issues and the theory for a fully automated approach based on causal dependencies.

Before working on a full implementation of our approach, we are currently testing the diversified instruction-set randomization technique we present in this article to determine its detection and tolerance capabilities. Then, we will consider extending our approach to data randomization [5, 20]: a technique which consists in diversifying the data by using different keys on each COTS server.

References


