Preventing malicious attacks by diversifying Linux shell commands*

Joni Uitto, Sampsa Rauti, Jari-Matti Mäkelä, and Ville Leppänen

University of Turku, 20014 Turku, Finland
{jjuitt, sjprau, jmjmak, ville.leppanen}@utu.fi

Abstract. In instruction set diversification, a "language" used in a system is uniquely diversified in order to protect software against malicious attacks. In this paper, we apply diversification to Linux shell commands in order to prevent malware from taking advantage of the functionality they provide. When the Linux shell commands are diversified, malware no longer knows the correct commands and cannot use the shell to achieve its goals. We demonstrate this by using Shellshock as an example. This paper presents a scheme that diversifies the commands of Bash, the most widely used Linux shell and all the scripts in the system. The feasibility of our scheme is tested with a proof-of-concept implementation. We also present a study on the extent of changes required to make all the trusted scripts and applications in the system use the new diversified shell commands.

Keywords: software security, instruction set diversification, Linux command shell, Bash

1 Introduction

In this paper, we present a diversification scheme which prevents the execution of undiversified command shell scripts in order to protect the system from malware. While the focus of our discussion is on injection attacks such as Shellshock, our scheme also generally prevents attacks in many situations where the attacker tries to execute a malicious script in the target system.

Among security bugs, vulnerabilities allowing code injection attacks are probably most commonly exploited by malware [3]. In these attacks, the code is inserted in the vulnerable program, enabling the attacker to use the program's privileges to launch an attack. Code injection attacks are known to be popular with binaries compiled from weakly typed languages like C, but are also often used to execute arbitrary code on other environments like SQL [19] or Unix command shell [13]. In this paper, we concentrate on preventing attacks in command shell environment.

Malware and attackers make use of the fact that the set of commands interpreted by the command shell is identical on each computer. Because of this

* The authors gratefully acknowledge Tekes – the Finnish Funding Agency for Innovation, DIGILE Oy and Cyber Trust research program for their support.

software monoculture, an adversary can design a single program that is able to successfully attack millions of vulnerable computers and devices. To defeat these kinds of attacks, we employ a method based on instruction set diversification.

The command sets of command shells on different computers, servers and devices can be uniquely diversified so that a piece of malware no longer knows the correct shell commands to perform a specific operation in order to access resources on a computer. As the malware is unfamiliar with the language used by the command shell, attempts to attack are rendered useless. Even if a piece of malware were to find out the secret diversified commands for one shell script, the same secret commands do not work for other scripts or systems. This diversification scheme can also be seen as proactive countermeasure against code injection attacks: The exact type of injection does not have to be known beforehand in order to thwart it.

It is also worth noting that diversification does not affect the software development process. The general idea in diversification (be it targeted at a command language or an API interface) is that software development is done against the ordinary reference language or API interface, and software artefacts are diversified machine-wisely after the development phase.

The contributions of this paper are as follows. We propose a scheme for diversifying Unix shell commands. Portokalidis et al. have briefly mentioned this idea in [15] among other possible applications of instruction set diversification. However, they do not go into much detail or provide any implementation for a diversified command shell. Our work can be seen as a continuation of this work, taking a more detailed and concrete approach on this issue. Our approach also significantly improves the security of their previous idea.

We present a proof-of-concept implementation of a diversified command shell, Bash, in order to demonstrate the feasibility of our approach in practice. We also show how our solution prevents code injection attacks, using different popular cases of Shellshock attack as examples. Additionally, we provide a brief study on the extent of changes required to make all the script files in two real life Linux distributions use the new diversified shell commands.

The rest of the paper is structured as follows. Section 2 describes the attack scenario. As an example, we describe Shellshock, an attack exploiting vulnerabilities in the Bash command shell and explain how our approach prevents this threat. In Section 3, we present our solution first as a general conceptual model and then as a practical implementation. Section 4 discusses the feasibility of shell diversification and presents some results on the number of the script files in two popular Linux distributions. The limitations of shell diversification are also covered. Section 5 contains the related work and Section 6 concludes the paper.

2 Attack scenario

Our solution aims to prevent the attacks where the attackers succeed to run malicious shell scripts or shell code fragments in the system. Code injection attacks are one typical way for adversaries to achieve this. Code injection usually happens against interfaces where the target system requests data from the user. If the system doesn't properly handle this data, it may become susceptible to code injection attack. Malicious user has an opportunity to offer the system data containing code instructions that could get executed.

For example, in C programming language, the function `int system(const char *command)` from `stdlib.h` runs the given command string as a shell command:

```
char command[100] = "ls -l ";
char *user_input;

/* ask a file name from the user here and put it in user_input */

strcat(command, user_input); /* add a file name to the input */
system(command); /* execute as a shell command*/
```

Now, if the user would give the string `"; cat /etc/passwd"` as an input, contents of the password file would be printed.

As another example of a possible attack scenario, we discuss Shellshock [5], a family of security bugs found in the widely used Unix Bash shell, first discovered on 24 September 2014. While the vulnerabilities making this attack possible have been patched, similar attacks are possible in future. Shellshock would have easily been defeated by our approach. Bugs like Shellshock are very critical, because many services on Internet, like several web servers, use Bash to process certain requests.

The Shellshock attacks made use of vulnerabilities in Bash, a program that several Unix-based operating systems utilize to run command scripts. Bash is often installed as the operating system's standard command-line interface.

In Unix-based systems, every running program possesses a list of environment variables, which are basically name-value pairs. When a running program invokes another program, it gives an initial environment variable list to this new process. In addition, Bash also internally stores a list of functions that can be run from within the program. When Bash invokes itself as a child process, the original instance can pass the environment variables and function definitions on to the new subshell. More specifically, the function definitions reside in the environment variable list as encoded variables, the values of which start with parentheses followed up by a function definition. When the subshell starts, it changes these values back into internal functions. The piece of code in the value is executed and a function is created dynamically on the fly.

The problem is that the Bash version vulnerable to the attack does not perform any check to make sure that the code fragment is a valid function definition. Attacker therefore has a chance to run Bash with a freely chosen value in its environment variable list. This means the adversary can execute any commands of his or her choice. Naturally, this arbitrary code execution would not be possible in the situation where the interpreter only accepts scripts conforming to the diversified script language.

As an example, Shellshock can be used to take control of a server. The following remote control attack attempts to use two programs – `wget` and `curl` – to connect to the attacker's server and download a program that the attacker can then use to control the targeted server [5]:

```
() { :;}; /bin/bash -c \"cd /tmp;wget http://213.x.x.x/ji;curl -O
/tmp/ji http://213.x.x.x/ji ; perl /tmp/ji;rm -rf /tmp/ji\"
```

The downloaded Perl program is run immediately and remote access for the attacker is established.

Attacks like Shellshock can potentially compromise millions of servers and other systems. However, if our implementation is in place, a successful attack requires knowing the diversified shell commands, that is, the secret used to diversify the original commands. Without this knowledge, many security vulnerabilities become useless. It follows that our solution is also proactive in the sense that it does not depend on the exact attack vector as long as the adversary tries to use a shell language to perform the attack. In what follows, we will provide a detailed description of our scheme.

3 Our solution

3.1 The conceptual diversification scheme

In our conceptual diversification scheme, a diversifier tool is used to produce uniquely diversified script files. These scripts can then be run only by an interpreter that supports diversified scripts. The interpreter executes the script by making use of the secret that has been used to diversify the script file. As the malicious adversaries do not possess the diversification secret, they cannot diversify their malicious code fragments correctly and their attacks are thwarted.

Our diversification scheme is shown in Figure 1. Each diversified script has its own secret that is used to generate the diversified script file with a diversifier tool and execute it with a diversified interpreter. The tokens in the scripts are diversified by combining the semantic value (that is, the string presentation) of the original token and a unique tag. In this context, ”token” means a collection of characters that is assigned a token identifier by the interpreter's lexer. The tag is calculated using the secret and the semantic value of the token under diversification (see the circles in Figure 1). Simple concatenation can be used but it is also possible to use some cryptographic function to combine these parts. For example, our implementation appends a hash value to the original token.

Our method only diversifies the tokens that occur in the script, so the adversary has no way of knowing the diversified forms of other tokens even if he or she somehow get access to the script's source code. It is worth noting that each token receives its own unique diversification. In this sense, we improve the

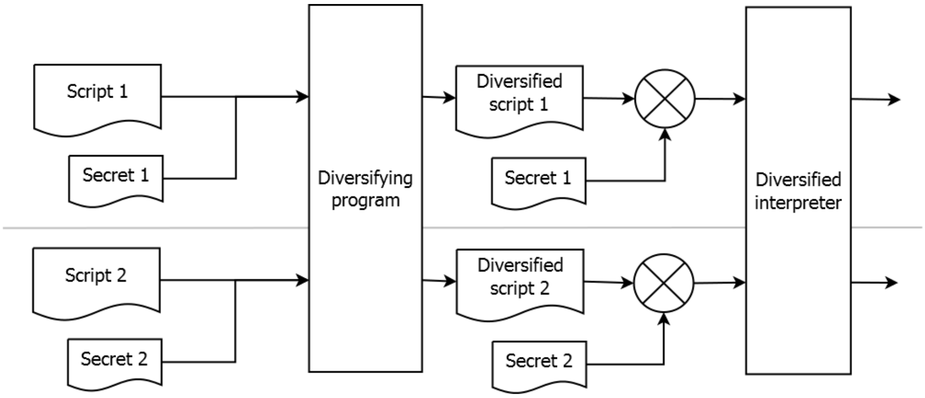

Fig. 1. Our conceptual diversification scheme.

solution suggested by Portokalidis et al. [15] where each token in a script file is diversified by appending the same secret tag to each token in a script file. Our scheme makes the diversification more secure by making the diversification of different tokens independent of each other. Taking this approach a step further, we can also vary the diversification of a token depending on the context it appears in. For example, the diversified form of a token can depend on preceding tokens or the location of the token in the script file. This makes it even harder for an attacker to guess the diversified forms of the tokens and inject anything into the script.

3.2 The practical implementation

Our proof-of-concept implementation of the diversified Bash shell was implemented by extending and modifying GNU Bash version 4.3.39. The implementation is written in C. The implementation and testing was performed using Ubuntu GNU/Linux 3.16.0-45-generic on 32-bit architecture. Our implementation itself is provided as an additional tool library, keeping direct modifications to the actual Bash interpreter minimal.

In [15], Portokalidis et al. implemented a proof-of-concept version of a Perl interpreter that executed Perl scripts with randomized instruction sets. The interpreter's lexical analyzer was modified to append a 9-number tag to each token recognized by the lexical analyzer. Our approach for diversifying Bash follows a similar design: we append a diversifying tag after each recognizable token's semantic value.

In Bash, these tokens can be keywords like `while`, `for`, `if` or more complex constructs like assignments such as `k=1`. As mentioned previously, the diversifying tag depends on the semantic value of these tokens. In Bash, the semantic value of the token `if` is "if" but for example `k=1` is understood as token of the type ASSIGNMENT_WORD, k=1 being its semantic value. Hence, the string k=1 receives a different diversifying tag from k=2 despite both being of the same token type.

The diversification process is shown in Figure 2. The diversification library provides an interface that the Bash interpreter uses during the tokenizing and execution phases. Each hash value is separated from a token with a distinct string of characters. This separator string is used to strip hashes from the input stream before it is passed to the lexical analyzer. For example, with the separator and a hash value, the echo command could become

```
echo~~~B2D21E771D9F86865C5EFF193663574DD1796C8F
```

After the lexical analyzer has determined which token it is currently handling, a hash is calculated for that token and compared with the collected hash. If the hashes match, execution is allowed. Otherwise, the diversified token is considered erroneous and execution of the script is halted.

The current implementation uses two different separators for the hashes. The first separator is meant for language specific reserved words and other tokens. The second separator informs the diversification library that the word before the separator should be a command word, that is, built-in utility function, a function call, or a program or script in the PATH variable. Before the command gets executed, it is parsed for a hash and it is compared to a hash calculated from the command word. As with other tokens, if the comparison is successful, execution is allowed, otherwise the script execution is halted.

In our proof-of-concept implementation of our diversified Bash interpreter, the hashes are generated using SHA-1. The hash is calculated by concatenating the original token and a token-specific secret. In [15] Portokalidis et al. included the secret in the beginning of the diversified script file or provided it as a command line argument to the interpreter. The secret was omitted from the executable script before parsing. Our solution currently uses the same approach but different methods of storing and handling the secret are quite easy to add.

As an example of script diversification, consider the following script that calculates a few first digits of the Fibonacci sequence:

```
Num=5
f1=1
f2=1
echo "The Fibonacci sequence for the number $Num is : "

for (( i=0;i<=Num;i++ ))
do
     /bin/echo -n "$f1 "
     fn=$((f1+f2))
     f1=$f2
     f2=$fn
done
```

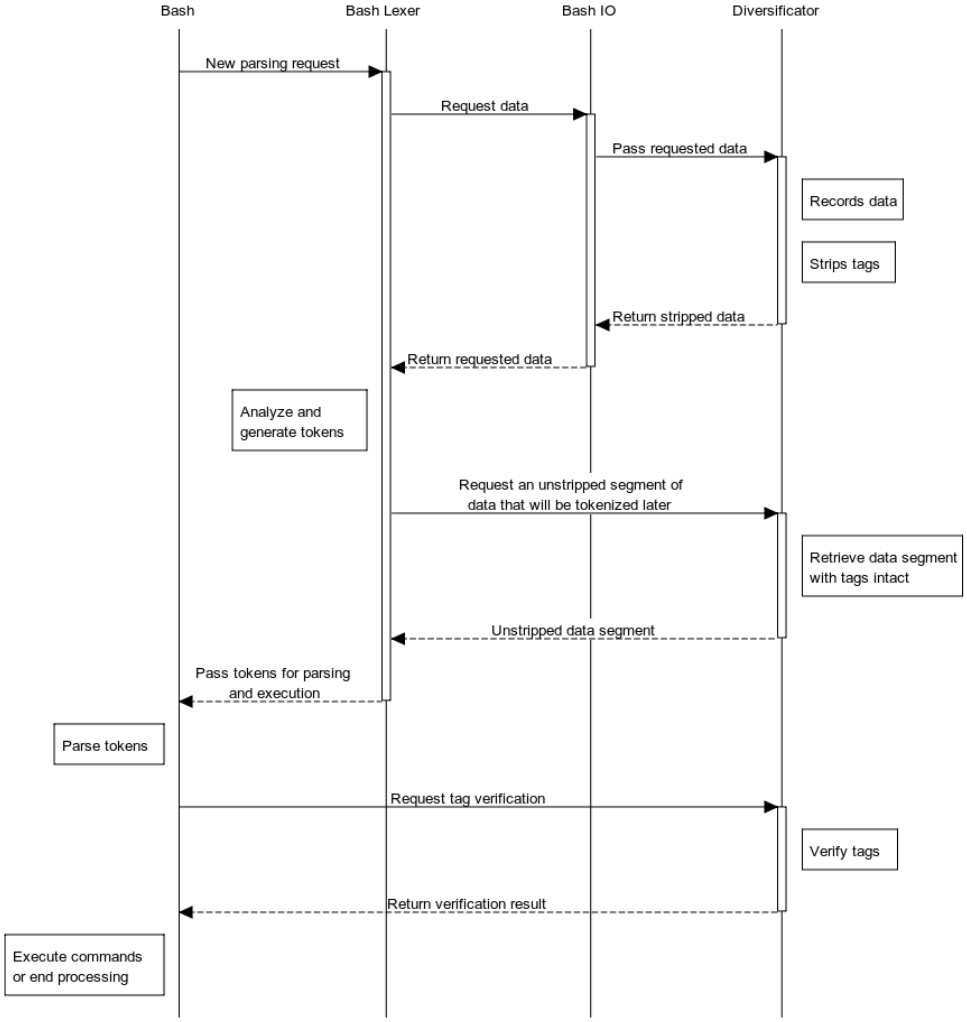
Bash
Bash Lexer
Bash IO
Diversificator
New parsing request
Request data
Pass requested data
Records data
Strips tags
Return stripped data
Return requested data
Analyze and generate tokens
Request an unstripped segment of data that will be tokenized later
Retrieve data segment with tags intact
Unstripped data segment
Pass tokens for parsing and execution
Parse tokens
Request tag verification
Verify tags
Return verification result
Execute commands or end processing

Fig. 2. The diversification process.

The diversified version of the script would look like the following:

```
Num=5^^^9D4D7FB947AFB1BA187FAEFB20533E918EE04212

f1=1^^^D0EE7568D8FE56441EA4BA60CEB119526C12CA06
f2=1^^^BB8630463671DBC49124A08566D6211B5BB90A6B

echo~~~B2D21E771D9F86865C5EFF193663574DD1796C8F
"The Fibonacci sequence for the number $Num is : "



for^^^D9000A6E1DBA2A95B2DDB13E74B220354B5B63AC
(( i=0;i<=Num;i++ ))^^^A04BDD7E8B4AB852FDC07FAF54E0107B12913976
do^^^23CF80A1D6201DAEA7112F6EA161DBA32A055BD2
   /bin/echo~~~BCD981E6B112655886C12639214C366EF6961F03 -n "$f1 "
   fn=$((f1+f2))^^^A52A61459E705054790329809CA21970B2999E77
   f1=$f2^^^451DBA3B0289063BCA2F6B7319D9F37F944C1BA6
   f2=$fn^^^7ECA3DF4236A6E384DE9ABABD46C4D53BEA2528A
done^^^14D13C75E6A9348DDD5561AD7F1155609175F38A
```

The hashes in this example, generated using SHA-1 function, are rather long and result into a considerable increase in source script file sizes. However, the module responsible for generating and validating the hashes can be easily extended to facilitate alternative methods of hash generation. For the sake of clarity, the hashes are encoded in hexadecimals in the previous example script. The test run performed on this diversified script and other similar examples executed without errors.

The purpose of our diversification library is to provide integrable diversification functionality with minimal changes required to the original interpreter. This would enable a multitude of Bash-like and other interpretable languages to be diversified relatively easily and lessen the burden of maintaining vastly different versions of diversifying script interpreters.

Integrating the diversification library into the existing Bash interpreter required fairly minimal changes to the Bash source code. Most changes were required in `parse.y`, the input file for the Bison parser generator. As mentioned before, the diversification module operates between Bash's I/O handlers and lexical analyzer. As Bash analyses the source code, the diversification module collects recognizable hashes for future comparison. When Bash's lexical analyzer identifies a token, diversification module catches these tokens and calculates hashes for them and compares them with the previously collected hashes. To make sure that code does not get executed before the tokens have been verified, the file `execute_cmd.c` was modified to ask permission from the diversification module to execute the parsed code.

3.3 Further notes on our approach

A big benefit of our approach is that it does not change the software development process. The programmer can write scripts as usual and the diversification of the script is performed by an automatic tool after the code has been written. The user experience is also not affected because the semantics of the scripts remain the same.

Diversification resembles encryption, and one might wonder why we do not encrypt the script files wholly in our scheme. There is a clear benefit in diversifying script languages instead of simply encrypting those script files. When executing an encrypted script, the file first needs to be decrypted. Once this step has been completed, the file is fed to the interpreter. Were an attacker to utilize an attack vector that would bypass the decryption phase entirely, such as a code injection attack, the system would remain vulnerable. In a code injection attack, the malicious code is placed inside the running program or script. This would circumvent the encryption-decryption process. Diversification prevents this scenario by renaming the language interface. Even if malicious code is injected in the running software, it will no longer match the language of the interpreter.

Moreover, unlike with completely encrypted code, with diversified code it is possible to use a renaming scheme in which the original command names are part of the diversified names. This way the code remains easily readable and also maintainable to some extent. The script could also be only partially diversified so that some parts of the code remain open to manual or automatic changes. In any case, it is worth noting that diversification and encryption can be used as separate layers of protection.

Security of our approach could also be futher improved with several methods. For example, the original language interface of the Bash command shell could be left in the system as a honeypot that catches malicious programs trying to use it. This is possible because no trusted program should use this original interface anymore. Other way to increase the resilience of our scheme is to make the diversification change dynamically over time. This way, the adversary will have much less time to figure out the diversification that keeps varying.

We also performed preliminary performance tests on our diversified interpreter. The test file consists of 5000 lines of randomized assignment operations. This file was then diversified in order to run experiments with our implementation. While the code example itself is naïve, it requires the diversified interpreter to undiversify each command. This represents the worst case performance scenario for our implementation. Many more complex command structures, such as loops, could be undiversified just once, even though their code is executed several times. The Both files were executed 100 times for both original and diversified Bash interpreter. The times were measured using Bash's built-in time-command. The standard Bash interpreter performed each execution at an average of 0.0164 seconds, while the diversified Bash performed at 0.0443 seconds. Hence, our diversified interpreter takes around 2.7 times longer to execute. Because we have not yet fully optimized our diversified interpreter and because the experiment was run using the worst case scenario, we do not consider this a large performance penalty.

4 Feasibility of shell diversification

In this section, we present a study of presence of script files in two Linux distributions and discuss some limitations of our diversification scheme.

4.1 A study of presence of script files in two Linux distributions

Our data was collected on Fedora 22 Server distribution and an older, minimal Gentoo distribution. More specifically, on Fedora, the command `uname -a` yields Linux 4.0.4-301.fc22.x86_64 #1 SMP Thu May 13:10:33 UTC 2015 x86_64 GNU/Linux. Respectively, Gentoo's `uname -a` is Linux Gentoo 3.14.4 #1 Tue May 20 11:04:51 EEST 2014 x86_64 GNU/Linux. During Fedora's installation process a few extra selections were made. The installation type Web Server was chosen and add-ons Tomcat, PHP and MariaDB were added to the installation to provide a touch of real-life server environment.

The process of cataloguing script files was performed using simple tools provided with the installation. First, qualifying files were aggregated using the `find` command and then filtered using a simple `grep` command. All commands were run with root privileges and only script files with execute permissions were searched for. A guard file was created in order to avoid files that are being actively updated. Finally, the `sed` command was used to remove a few pure binary files from the results:

```
# touch guard
# find / -type f -perm /a+x ! -newer guard > files
# xargs grep ’^#![/a-z]*/bin/[a-z0-9]*’
  < files > grepmatches 2> /dev/null
# sed -i ’/Binary/d’ grepmatches
```

The results of this process were processed with a simple Python script. Interpreter paths of the form #!/usr/bin/env X were shortened to either #!/usr/bin/X or #!/bin/X where appropriate. The first column of Tables 1 and 2 shows the interpreter referenced by the script on the shebang (#!) line and the second column has the number of such references. Due to the grepping procedure, some files were listed twice. Those files were eliminated in post-processing.

In addition to executable scripts, we also aggregated non-executable library scripts by first listing all files in the system using

```
# find / -type f
```

These files were then filtered using the command

```
# grep grep ’\.py[co]*$\|\.sh$\|\.p[lm]$’ files > libraryfiles
```

The filtering process relies upon file extensions used by library developers. In Unix-based systems there is no guarantee that file names contain a file extension. However, most well maintained libraries adhere to the convention of using file extensions and thus the numbers give an accurate enough estimation on the quantity of scripts in a fresh system installation.

The data we collected on executable and non-executable scripts were combined using a simple Python script. This script ensured that every file would be calculated only once (having a file extension and a shebang would qualify the file for both executable and non-excutable categories). Some libraries have multiple versions of the same file, for example, python library might include `script_file.py`, `script_file.pyo` and `script_file.pyc` where the first file is the source file and the two latter files are byte-code files. In this case `script_file` would only be added to the sum once.

The script files – both executable and non-executable – found in Fedora and Gentoo are shown in Table 1 and Table 2, respectively. Comparing these two tables, we see Fedora has 2319 script files more, but it is also a bit more service-oriented distribution. The biggest difference seems to be in the number of library scripts, Gentoo has more Perl, Python and shell libraries than Fedora. Other than that, the number of scripts is quite similar. The installations themselves are also fairly similar in size (about 80 MiB).

What can be deduced from this data, then? There are quite many script files in both distributions we studied. Still, diversifying them all would not be a huge work for an automated diversifier. It is also worth noting that most of the scripts are Bash or sh scripts that can be handled (sh is a subset of Bash and diversified sh scripts can therefore be run using our implementation). Perl and Python scripts also seem to make up a significant proportion of all the scripts in the system, so covering the interpreters of these script languages would be important for a comprehensive script diversification system. Also, some of the scripts can be rewritten to use a different interpreter to achieve a completely diversified solution.

4.2 Limitations of shell diversification

Although our diversification scheme provides many advantages from the security point of view, it also has some limitations and drawbacks. Obviously, diversifying all scripts in the system introduces a problem for users who want to use the command shell manually. After all, it would be too laborious for the users to write diversified keywords and scripts. We could provide users with a separate terminal for inputting shell commands, but this solution can be a security risk as the malware may find a way to use this interface as well. On the other hand, many normal users are not able or do not need to use the command shell. In some remote systems, the need for an interactive local shell could be replaced by remote administration tools.

Another challenge is the problem of diversifying all the scripts and programs that may dynamically create new scripts at runtime. Still, an automatic diversifier program that programmers can use when adding scripts to their programs

Table 1. Script files found in Fedora.

Interpreter path	**Number of files**
#!/bin/bash	154
#!/bin/sh	349
#!/usr/bin/bash	3
#!/usr/bin/lua	1
#!/usr/bin/perl	50
#!/usr/bin/python	75
#!/usr/bin/python2	2
#!/usr/bin/python2.7	1
#!/usr/bin/python3	10
#!/usr/bin/python3.4	2
perl libraries	1085
python libraries	3705
shell libraries	31
Total	**5469**

Table 2. Script files found in Gentoo.

Interpreter path	**Number of files**
#!/bin/bash	149
#!/bin/csh	1
#!/bin/sh	237
#!/usr/bin/awk	3
#!/usr/bin/jimsh	1
#!/usr/bin/lua	1
#!/usr/bin/perl	107
#!/usr/bin/perl5.16.3	1
#!/usr/bin/python	40
#!/usr/bin/python2	2
#!/usr/bin/python2.7	15
#!/usr/bin/python3	6
#!/usr/bin/python3.3	13
#!/usr/bin/bash	3
perl libraries	1972
python libraries	4991
shell libraries	251
Total	**7788**

can be created for this purpose. This way, the programmer does not have to manually diversify any scripts that might be included in his or her program code. Also, minimal systems – for instance the operating systems for IoT devices – contain much smaller amounts of script files and scripts included in program code and are thus easier to handle with regard to our approach.

Installing new programs and scripts into the system can also introduce some problems. In order to work correctly, new programs and scripts also need to be diversified using a diversificator tool. In some systems – such as small-scale systems on IoT devices – the issue could be mitigated by preferring image based full-system updates over in-place updates of system files run by scripts. Therefore, at least for minimal and restricted IoT environments, our solution can be expected to work well. In the IoT context, the size of the full system may only be a few megabytes, which makes it quite easy to apply the diversification and fix possible issues.

In instruction set diversification schemes in general, storing the secret diversification key or keys securely is also an issue. In our scheme, however, we assume the attacker does not have an access to the file system of the computer he or she is targeting; this is the case in Shellshock and similar attacks in which the adversary is trying to gain an access to the system. Therefore, for the purposes of this attack scenario the file system can be seen as a safe place to store the secret key. Of course, some stronger cryptographical storing schemes could also be considered.

5 Related work

Instruction set randomization has been applied to several different software layers and many different application areas. Portokalidis et. al present the model closest to our work in [15]. The authors apply instruction set randomization to a Perl interpreter, enabling execution of diversified scripts. Perl code injected by an adversary will fail to run because it is not correctly diversified and is not recognized by the system. They also briefly mention the idea of diversifying shell scripts but do not provide any details or implementation. In this sense, our work can be seen as continuation for their paper.

Boyd and Keromytis have studied the SQL language [4]. Their intermediate proxy, SQLrand, translates the diversified queries into the original SQL language and passes them on to the database. We present an improved scheme and implementation for SQL randomization in [19]. Many papers [3, 15] and books [9, 10] also study the idea of system-wide, global instruction set diversification. Building diverse operating systems and software systems in general has been suggested by Cohen already in the nineties [6]. Forrest [8] also discusses diversified software systems as a security measure.

Barrantes et al. have studied instruction set randomization on binary level to defend against code injection attacks [1, 2]. There has also been some interest in randomizing the system call numbers to render malicious code useless: Jiang et al. [12] and Liang et al. [14] have studied this issue. We have also presented a tool for system call randomization [16]. Using similar ideas, randomization of memory addresses has been used to prevent memory exploits [7]. The common factor for all of these approaches is that the basic idea is to change the language of the system in order to prevent malicious programs from using some kind of interface that provides access to a resource.

Identifier obfuscation scrambles identifier names on source code level. This is conceptually somewhat similar to our diversification scheme and has been discussed in many papers. For example, there are diversifying tools for Java [20] and JavaScript [11]. We have studied this topic and built a tool that scrambles identifiers and function signatures in web applications written in JavaScript and HTML [17, 18].

6 Conclusions

We presented an instruction set randomization based scheme for preventing code injection attacks in Linux shells. By diversifying the tokens of the Bash scripts uniquely, we prevent the attacker from possessing the knowledge about the correct script language beforehand. We have also discussed the practical implementation for our scheme and explained the effectiveness of our scheme against Linux shell code injection attacks such as Shellshock. We also discussed how our solution improves security over a previous diversification approach.

A study of the presence of script files in two popular Linux distributions was also presented. Based on this, it seems that Perl and Python interpreters should also be covered in a practical and comprehensive script diversification scheme. Therefore, possible future work includes developing our diversificator library to more general direction in order to handle other script languages like Perl and Python.

One limitation of our approach is that all scripts in a system need to be diversified. However, this is quite possible at least in many restricted server environments and small IoT environments with a limited number of scripts and infrequent updates. Also, the diversification could be performed automatically for the most part.

References

1. E.G. Barrantes, D.H. Ackley, S. Forrest, and D. Stefanovic. Randomized Instruction Set Emulation. *ACM Trans. Inf. Syst. Secur.*, 8(1):3–40, 2005.
2. E.G. Barrantes, D.H. Ackley, T.S. Palmer, D. Stefanovic, and D.D. Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 281–289, 2003.
3. S.W. Boyd, G.S. Kc, M.E. Locasto, and A.D. Keromytis. On the General Applicability of Instruction-Set Randomization. *IEEE Transactions on Dependable and Secure Computing*, 7(3), 2008.

4. S.W. Boyd and A.D. Keromytis. SQLrand: Preventing SQL Injection Attacks. In *Applied Cryptography and Network Security*, Lecture Notes in Computer Science Volume 3089, pages 292–302, 2004.
5. CloudFlare. Inside Shellshock: How hackers are using it to exploit systems. Available at: https://blog.cloudflare.com/inside-shellshock/, 2014.
6. F.B. Cohen. Operating System Protection through Program Evolution. *Comput. Secur.*, 12(6):565–584, 1993.
7. D.C. DuVarney, V.N. Venkatakrishnan, and S. Bhatkar. SELF: A Transparent Security Extension for ELF Binaries. In *Proceedings of New Security Paradigms Workshop*, 2003.
8. S. Forrest, A. Somayaji, and D. Ackley. Building Diverse Computer Systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, HOTOS '97, 1997.
9. S. Jajodia, A.K. Ghosh, V.S. Subrahmanian, V. Swarup, C. Wang, and X.S. Wang. *Moving Target Defense II, Advances in Information Security 100.* Springer, 2013.
10. S. Jajodia, A.K. Ghosh, V. Swarup, C. Wang, and X.S. Wang. *Moving Target Defense, Creating Asymmetric Uncertainty for Cyber Threats, Advances in Information Security 54.* Springer, 2011.
11. Q. Jiancheng, B. Zhongying, and B. Yuan. Polymorphic Algorithm of JavaScript Code Protection. In *Proceedings of International Symposium on Computer Science and Computational Technology, ISCSCT '08*, pages 451–454, 2008.
12. X. Jiang, H.J. Wang, D. Xu, and Y-M. Wang. RandSys: Thwarting Code Injection Attacks with System Service Interface Randomization. In *IEEE International Symposium on Reliable Distributed Systems*, SRDS 2007, pages 209–218, 2007.
13. G.S. Kc, A.D. Keromytis, and V. Prevelakis. Countering Code-injection Attacks with Instruction-set Randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 272–280, 2003.
14. Z. Liang, B. Liang, and L. Li. A System Call Randomization Based Method for Countering Code Injection Attacks. In *International Conference on Networks Security, Wireless Communications and Trusted Computing*, NSWCTC 2009, pages 584–587, 2009.
15. G. Portokalidis and A.D. Keromytis. Global ISR: Toward a Comprehensive Defense Against Unauthorized Code Execution. In *Moving Target Defense, Creating Asymmetric Uncertainty for Cyber Threats, Advances in Information Security 54*, 2014.
16. S. Rauti, S. Laurén, S. Hosseinzadeh, J. Mäkelä, S. Hyrynsalmi, and V. Leppänen. Diversification of System Calls in Linux Binaries. In *To be published in proceedings of the 6th International Conference on Trustworthy Systems (InTrust 2014)*, 2014.
17. S. Rauti and V. Leppänen. A Proxy-Like Obfuscator for Web Application Protection. *International Journal on Information Technologies & Security*, 5(1), 2014.
18. S. Rauti and V. Leppänen. Man-in-the-Browser Attacks in Modern Web Browsers. In *Emerging Trends in ICT Security*, 2014.
19. S. Rauti, J. Teuhola, and V. Leppänen. Diversifying SQL to Prevent Injection Attacks. To be published in proceedings of International Conference on Trust, Security and Privacy in Computing and Communications, 2015.
20. T. Zhangyong, C. Xiaojiang, F. Dingyi, and C. Feng. Research on Java Software Protection with the Obfuscation in Identifier Renaming. In *Fourth International Conference on Innovative Computing, Information and Control (ICICIC)*, pages 1067–1071, 2009.