Internal Interface Diversification with Multiple Fake Interfaces *

Sampsa Rauti
University of Turku
sjprau@utu.fi

Ville Leppänen
University of Turku
ville.leppanen@utu.fi

ABSTRACT

Malware uses knowledge of well-known interfaces to achieve its goals. However, if we uniquely diversify these interfaces in each system, the malware no longer knows the "language" of a specific system and it becomes much more difficult for malicious programs to operate. This paper extends the idea of interface diversification by presenting a scheme where a fake original interface and multiple other fake interfaces are provided along with the valid interface in order to log the suspicious activity in the system and possibly deceive malware by initiating fallacious interaction with it. We also present a proof-of-concept implementation of this scheme in Linux environment and conduct experiments with it.

Keywords

Diversification, Moving target defence, Deception

1. INTRODUCTION

Protecting computer systems from attacks requires taking into account the complex environment with several previously unknown security holes. On the other hand, a malicious attacker only needs one vulnerability in order to compromise a system. However, malware often relies on prior knowledge on known interfaces – most attacks depend on the known internal interfaces provided in the target system [17].

Due to the software monoculture prevailing today, there are huge number of instances of the same execution platform having identical internal interfaces. A malicious exploit therefore works on millions of systems. If the internal interfaces that malware uses for accessing the essential resources of a system would be unique on each computer, exploits relying on the knowledge about the identical interfaces would become useless (see e.g. [3] and [6]).

We refer to this scheme as *interface diversification*. In a diversified system, malware does not possess the knowledge of the secret "language" used in the system and is not able to access any important services and resources offered by the system. The trusted programs, on the other hand, are diversified accordingly in order to be compatible with the diversified interfaces they utilize.

The contributions of this paper are as follows. We extend the idea of internal interface diversification by introducing the concept of multiple fake interfaces and constantly changing diversification. The approach we present can be seen as combining the ideas of internal interface diversification [9, 12], fake interfaces [11] and moving target defense [8, 14].

The rest of this paper is organized in the following way. Section 2 explains the general idea of internal interface diversification. Section 3 extends this idea to multiple fake interfaces and a dynamically changing valid interface. Section 4 discusses how fake interfaces can be used to monitor and deceive malware. Section 5 presents our proof-of-concept implementation in Linux environment and related experiments. Section 6 discusses internal interface diversification with multiple fake interfaces as a security measure, and describes the strengths and drawbacks related to multiple fake interfaces. Benefits and drawbacks of the scheme are also discussed. Finally, Section 7 concludes the paper.

2. INTERFACE DIVERSIFICATION – THE GENERAL IDEA

When we apply internal interface diversification, we limit the number of assumptions an attacker is able to make about the target system's internal interfaces. The interfaces are altered to make them unpredictable for malicious adversaries. In practice, diversification can be implemented by using different obfuscation techniques [4]. In the context of interface diversification, even simple obfuscation transformations like renaming functions and altering the order of parameters in their signatures can be employed.

By *interface*, we mean anything that can potentially be used to gain access to essential resources of a computer. Therefore, not only ordinary interfaces provided by libraries and software modules but also for instance commands of a shell language or even memory addresses (of specific services or resources) are considered as interfaces that can be diversified to protect the system from malicious attacks.

An example that illustrates internal interface diversification in practice is altering the mapping of system calls of

*The authors gratefully acknowledge the support of The Scientific Advisory Board for Defence (MATINE). This research is also funded by Tekes – the Finnish Funding Agency for Innovation, DIMECC Oy and CyberTrust research program.

ECSA, November 28-December 02, 2016, Copenhagen, Denmark

 ISBN 978-1-4503-4781-5/16/11…$15.00
DOI: http://dx.doi.org/10.1145/2993412.2993417

an operating system. Applications request services from the operating system's kernel using system calls in order to access the computer's resources.

In the Linux operating system, for instance, diversification can be achieved by replacing the original system call numbers with new ones. Diversification also has to be propagated to all pieces of code that invoke the system calls. Respectively, the names of library functions that directly or indirectly issue system calls have to be changed so that the malicious programs can not use them. That is, the transitive closure of the system calls is diversified as a part of the scheme so that harmful programs cannot exploit any system call entry points to use the resources provided by the system. Moreover, since the idea here is to diversify the system call numbers in each system uniquely, the adversary cannot use the knowledge gained by compromising one system to launch a large-scale attack covering other systems.

Note that internal interface diversification does not prevent the malware from entering the system, it just renders useless malicious programs' attempts to use the resources of the system. Another thing worth noting is that the diversification does not cause extra work for developers. Interface diversification can be done with an automatic diversifier tool after an application or a library is ready for deployment. Diversification is also invisible to the users of the system. This is because we do not target external interfaces visible to user but diversify internal interfaces that are usually used by adversaries.

3. INTERFACE DIVERSIFICATION USING MULTIPLE FAKE INTERFACES

In this section, we discuss interface diversification with multiple fake interfaces and several related approaches for extending and strengthening interface diversification as a security measure. These include e.g. continuously changing diversification and finer diversification granularity.

3.1 The Idea of a Fake Original Interface

Applying diversification does not necessarily have to mean that we want to completely prevent the malware from operating. If we also want to understand the behavior of malware, we could present it with a fake interface providing the functionality of a real target environment. However, the fake interface would limit malware's ability to cause damage to the infected system and prevent it from spreading to the other hosts in a network.

In order to achieve this, we can use a *fake original interface*. For example, using the system call interface as an example, we have the real diversified interface used by the trusted programs, but also leave the the original system call numbers in the system so that malware can use them (without any real adverse effects). This fake original interface makes it possible to observe and log malware's activities. Moreover, it is possible to deceive a malicious program into believing its operations are successful in the target system. This paper concentrates on the fake interfaces but does not deal with building further fake resources or bogus data used for deception. In order to deceive malware, it is clear that we would need a honeypot solution that keeps pretending that the actions of malware really have some effects on the system. We have addressed this kind of advanced deception elsewhere [15].

With the diversified interface and fake original interface in place, each system service will be accessible using the diversified secret system call number and the original interface exists as a kind of a honeypot. All the applications, services and libraries in the system are made compatible with the diversified system call interface. Consequently, trusted code should never use the original interface. Malware that uses the well-known system call interface directly can now be easily detected. A process invoking a well-known system call is always unexpected and suspicious – in these cases something unwanted is definitely executing in the system. This way, diversification can be combined with intrusion detection and analyzing behavior of malware.

Although the system call interface is used as an example above, diversifying it is not enough. This is because the applications often use wrapper functions when they need to invoke the system's services. Several wrapper functions available in the C standard library are examples of such functions. Therefore, we also have to diversify all the entry points leading to system calls to effectively prevent malicious pieces of code from causing harm in the system.

3.2 Multiple Fake Interfaces

The scheme described above can be extended. We can make it significantly more difficult for malware to break the diversification scheme by introducing multiple diversified interfaces. One of these interfaces at a time is the *valid interface* that can be utilized by applications to use the services provided by the system. Other interfaces are fakes, possibly logging the suspicious activity in the system.

The basic idea of multiple fake interfaces is shown in Figure 1. There is the fake original interface which always remains a fake and a set of other interfaces, only one of which is the valid interface at a time. Each trusted program knows how to use this valid interface, but a malicious program either uses the fake original (malware 1 in the figure) or maybe if its more sophisticated and has an ability to perform dynamic analysis, finds a way to use some of the fake interfaces (malware 2 in the figure).

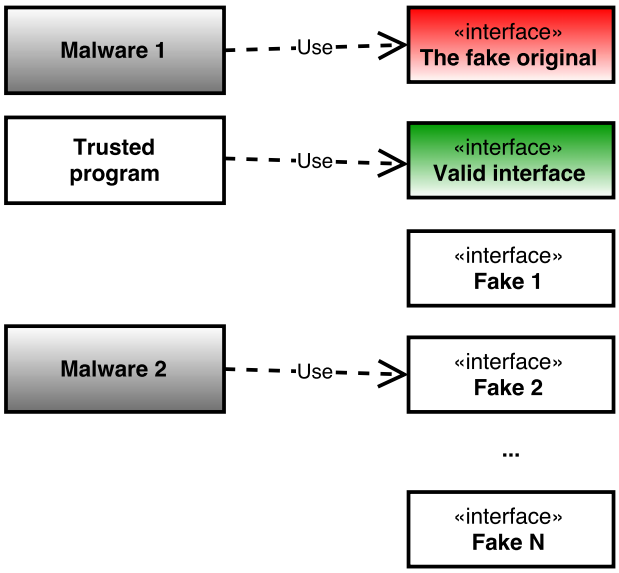

Figure 1: The idea of multiple fake interfaces.

Multiple fake interfaces can be used to strengthen the diversification considerably. When this method is applied to the system call interface, for example, it is not enough for a malicious program to guess only one group of system calls, because one set of these calls only remains valid for a very short time as the mapping is continuously changed.

If we only have one secret diversified interface that does not change and a malicious program somehow manages to dynamically analyze the calls issued by a trusted program, malware may succeed in deducing the meanings of diversified calls and cause harm in the system (this should not be easily possible, though, because such analysis would usually require using system calls, which is exactly the thing malware is prevented from doing here). This scenario can be made much more unlikely and more difficult for the malware by continuously changing the valid internal interface used by an application (e.g. changing it after each call). In other words, our scheme considerably complicates the dynamic analysis that would be necessary to figure out the secret mapping.

3.3 Dynamic Diversification

As became apparent above, the correct interface with the genuine access to the resources of the computer should be continuously changed to make it harder for the malware to utilize resources successfully. This idea bears a resemblance to continuously changing obfuscation introduced by Hohl [7] and Collberg [5]. However, unlike in these schemes, we propose switching the valid interface, not replacing chunks of program code on the fly.

An important question is how often the valid interface should be changed. There are several options:

- *Boot time re-diversification.* Change the valid diversified interface on each reboot of the system. This is the least secure option but incurs practically no performance penalty related to changing the valid interface.
- *Load time re-diversification.* Switch the valid diversification at load time. This option is only applicable when a separate diversified interface is being used for each application or library.
- *Run-time re-diversification.* Change the valid interface on the fly all the time (either in the whole system or for applications separately). This means the valid diversified interface keeps changing at runtime while several processes are being executed in the system. This approach is effective even when the malware performs dynamic analysis at runtime.
- *Re-diversification after every call.* Change the valid interface after some of its entry points is invoked. This alternative prevents a replay attack where the malware would somehow be able to immediately mimic a call made by a trusted application. Re-diversification after every call could also be be combined with the previous option, continuous runtime re-diversification.

3.4 Diversification Granularity

In any diversification scheme, granularity is an important factor. Diversification granularity determines how large part of the system uses one diversified interface at time. It is possible to have one valid interface in the whole system at a time, one separate valid interface for each application or library, or even several valid interfaces for parts of one executing process. More specifically, the alternatives include:

1. Use coarse – and potentially less secure – diversification granularity. For example, one system-wide diversification for all the applications and libraries in the system. That is, all applications have the same valid interface at each moment.
2. Modify the system so that each application has a separate valid interface among the set of interfaces. Implementation and operation costs become higher. However, the diversification of each application is now isolated and some critical parts of the system can be re-diversified more often than some other parts.
3. Make diversification depend on location of invocation in the program code or memory [13, 16]. This means the same entry point is diversified differently in different positions in the same application or library. Different parts of the same program would have to call a different valid interface at a specific point in time. This option provides the finest granularity but is also quite complex.

Making use of fine-grained granularity along with multiple fake interfaces provides good security, but the performance and space overheads are higher. However, even the system-wide diversification should be quite a safe solution if the correct diversified interface is changed regularly.

4. MONITORING MALICIOUS BEHAVIOR AND DECEIVING MALWARE

When we monitor the suspicious calls made by malware, our fake interface scheme can be viewed as an anomaly-based intrusion detection system. The idea of anomaly detection systems is to separate abnormal behaviors by using a set of rules to classify observed actions [2]. The observed behavior is classified either as normal or abnormal behavior for a system. In our scheme, for example, a system call made using the well-known system call interface – or any other interface that is not currently the valid interface – is seen as a signal that uninvited code is present in the system. The problem of false positives has usually plagued anomaly-based detection schemes [19]. An important advantage of our dynamic fake interface approach is that it does not produce false alarms. At the same time, there is a risk that our system lets some malicious activity go undetected.

Of course, the interesting question is what kind of action should be taken after we detect anomaly in the system. In their simplest form, honeypot systems can simply raise an alarm on any unwanted attempt to access the system's resources. The suspicious activity is logged and the system's administrator is notified about the incident. Also, the system can automatically stop the execution of the targeted process and do something about the malicious program.

However, if we want to build a honeypot system to study behavior of malware, instead of immediately raising an alarm we can try to interact with malware in order to learn about its activities. That is, when the malicious program invokes some of our fake interfaces, we can give it fake response and fallacious data to deceive it into thinking its actions have

a real effect in the system while in reality this is not the case. For this purpose, we could have different kinds of *fake entities* planted in the system [15].

The goal here would be to attempt to simulate a normal conversation between malware and the system as well as we can, deceiving malware and collecting information about its behavior at the same time. A framework with this kind of deception and behavioral monitoring could even form a basis for learning whether the malware somehow tries to combat or circumvent internal interface diversification and multiple fake interfaces.

5. A PROOF-OF-CONCEPT IMPLEMENTATION AND EXPERIMENTS

To demonstrate the feasibility of the idea of multiple fake interfaces and a dynamically changing valid interface, we built a proof-of-concept implementation for Linux environment. This section presents our experimental implementation and offers some observations on the experiments we performed.

5.1 Implementation

Our experimental proof-of-concept implementation was built in C++. The objective was to test the idea of multiple interfaces and a changing valid interface by using a part of GNU C library (glibc), an implementation of C standard library, as an original interface. More specifically, for simplicity we chose 20 system call wrapper functions as an interface for which we built a fake original interface and a set of fake interfaces. Replacing the rest of glibc would be an identical task.

C standard library is an obvious choice to apply our approach to, because the use of any critical resources of the system (creating processes, managing memory, deleting files etc.) happens through this library. Malware can also be expected to take advantage of this library if it is not using the system call interface directly.

The attack scenarios our proof-of-concept implementation addresses cover the threats where the malicious program tries the invoke functions of a library using their well-known names and also the cases where the adversary finds out the secret valid interface by dynamically analyzing the behavior of a trusted program and then tries to invoke a function in this interface (which no longer belongs to the currently valid interface by that time). The first scenario is prevented because the original functions no longer work and the second scenario is made considerably more unlikely to work because of the very limited time window the attacker has.

Our implementation also acts as a simple monitor of malicious activity. Whenever the fake original interface or some of the current fakes is used, the behavior (the called function, given parameters and time) is logged, but the excepted original operation is not performed.

The implementation can be divided in three parts. First, there is the modified application that issues calls to the interfaces. Second, embedded in application there is diversifying functionality that constantly changes the function pointers in the application so that they refer to the valid interface. For example, function pointers are changed so that all the calls to `exit()` system call wrapper are redirected to a specific function implementing this functionality in the valid interface. Third, we have the modified glicb library containing the set of interfaces (one valid and multiple fakes) and the fake original interface. The implementation is depicted in Figure 2.

The function pointers in the application are constantly changed (by a function that maps the current time stamp to the function names in the valid interface). To avoid the problematic cases where a function call might fail because it is made just before the valid interface changes, the periods of validity for the previous and the new valid interface have to overlap for some time (at least few dozens of milliseconds). This takes care of potential synchronization problems between web application and interfaces.

5.2 Experiments

...

5.3 Limitations

There are a couple of limitations in our approach. First, there is always a possibility a malicious program is able to transfer its execution to a point in our program that allows it to call the current valid interface. Alternatively, the malicious program can try to analyze the engine responsible for changing the function pointers that is embedded in the program. However, both of these threats can be mitigated with the encryption approach we mentioned in Section 3.2 above.

Also, we naturally also need a diversifier tool that will automatically add the diversification engine to trusted programs and identify and modify the points in the program that call the diversified interface. This was not implemented as a part of this study since our purpose here was only to provide a proof that multiple interface diversification can be made to work smoothly without any problems in the execution of the program or large performance penalties. However, adding the diversifying engine to the program and identifying function calls is doable as a part of the compiling process or by employing binary rewriting as demonstrated in our earlier study [10].

While our experiments show that diversification with multiple fake interfaces is possible and a promosing approach, experiments in a larger scale are still necessary to further validate the practicality of the scheme.

6. DISCUSSION

Currently, information technology systems are designed to function with fixed configurations. Several kinds of identifiers like names, addresses and configuration parameters remain unchanged over long periods of time. Internal interface diversification with multiple fake interfaces can be seen as a breed of Moving Target Defense that alleviates the problems caused by this traditional static approach.

By continuously changing the valid internal interface, we increase the complexity and uncertainty for the adversary, reduce his or her window of opportunity and increase the costs caused by their efforts. When the exposure of software vulnerabilities and attack opportunities is limited and the resiliency of the system system increased.

Internal interface diversification is a general approach in the sense that it can be applied to any interface in the system. Examples include system call interface [13], operating system libraries [10], command shell languages [9, 20], database query languages [1] and memory layout [18]. The approach is also orthogonal: other traditional security mea-

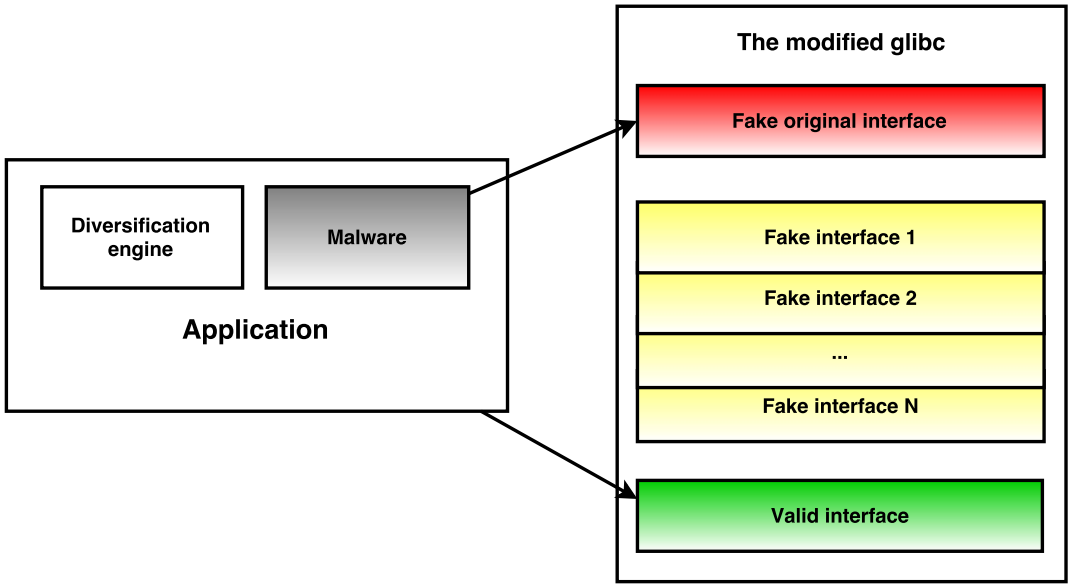

Figure 2: A rough diagram depicting the idea of our JavaScript implementation.

sures can still be used along with it. This is important as internal interface diversification does is not meant prevent the malware from infiltrating the system. Instead, it prevents the malware from working and allows the collection of data to analyze its behavior. Proactiveness is also an important feature: it does not need to know the nature or objectives malware beforehand, so the analysis and prevention of previously unknown is also possible.

Internal interface diversification with multiple interfaces also causes some challenges that have to solved. For example, storing entry point mappings for several different interfaces takes some extra space. Still, keeping track on simple mappings like system call numbers or function names does not usually consume that much space. Having several different interfaces for each program obviously requires more disk and memory space. We can alleviate the space consumption by generating more interface replicas for critical programs and libraries and less for other ones. In case of relatively sizable libraries, we can also use more lightweight versions of them. For example, uClibc is a minimal version of glibc, the C standard library. glibc is so large because it is intended to support all popular C standards across a wide spectrum of different hardware and kernel platforms. uClibc is mainly meant for embedded Linux and the included features can be chosen based on space requirements. Moreover, if we assume the system includes an automatic diversifier, new interfaces can also be created on the fly and old ones can be replaced by them, which saves lots of space. In this case, however, special care must be taken not to expose the diversification engine to malware.

Performance is another issue. While we have seen in our experiments that overhead is not unreasonable by any means, constantly changing the valid interface and the processes using it does consume processor time. In the end, this is a typical trade-off between performance and security. As we saw in Section 3.3, in many cases we may settle for significantly less performance-intensive models like load time diversification.

Taking into account these space and performance considerations, interface diversification with multiple fake interfaces should also prove to be a good fit for IoT operating systems if disk space is used economically and diversification is not changed so often that it becomes a burden to good performance.

7. CONCLUSIONS

This paper has extended internal interface diversification by introducing an approach where a fake original interface and multiple other fake interfaces are used in addition to a constantly changing valid interface. The proposed approach reduces the window of opportunity for adversaries and also enables us to log malicious activities in the target system.

As a future work, it would be interesting to build proof-of-concept implementations for environments other than web as well. Reducing disk space usage and optimizing performance of the approach are also topics worthy of further study. We believe using multiple fake interfaces is a worthwhile approach for strengthening diversification and improving software security while at the same time enabling intrusion detection and malware analysis.

References

[1] S. Boyd and A. Keromytis. SQLrand: Preventing SQL Injection Attacks. In *Applied Cryptography and Network Security*, Lecture Notes in Computer Science Volume 3089, pages 292–302, 2004.

[2] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3):15:1–15:58, July 2009.

[3] F. Cohen. Operating System Protection through Program Evolution. *Comput. Secur.*, 12(6):565–584, Oct. 1993.

[4] C. Collberg, C. Thomborson, and D. Low. A Taxonomy of Obfuscation Transformations. Technical Report 148, The University of Auckland, 1997.

[5] C. Collberg, S. Martin, J. Myers, and J. Nagra. Distributed application tamper detection via continuous software updates. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 319–328. ACM, 2012.

[6] 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.

[7] F. Hohl. Time limited blackbox security: Protecting mobile agents from malicious hosts. In *Mobile Agents and Security*, pages 92–113. Springer-Verlag, 1998.

[8] S. Jajodia, A. K. Ghosh, V. Swarup, C. Wang, and X. S. Wang. *Moving target defense: creating asymmetric uncertainty for cyber threats*, volume 54. Springer Science & Business Media, 2011.

[9] G. Kc, A. 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.

[10] S. Lauren, P. Mäki, S. Rauti, S. Hosseinzadeh, S. Hyrynsalmi, and V. Leppänen. Symbol diversification of Linux binaries. In *2014 World Congress on Internet Security (WorldCIS)*, pages 74–79. IEEE, 2014.

[11] S. Laurén, S. Rauti, and V. Leppänen. An interface diversified honeypot for malware analysis. In *Proccedings of the 10th European Conference on Software Architecture Workshops, Copenhagen, Denmark, November 28 - December 2, 2016*, pages 29:1–29:6, 2016.

[12] S. Laurén, P. Mäki, S. Rauti, S. Hosseinzadeh, S. Hyrynsalmi, and V. Leppänen. Symbol diversification of linux binaries. In *World Congress on Internet Security (WorldCIS-2014)*, pages 74–79. IEEE, 2014.

[13] 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.

[14] V. Pappas, M. Polychronakis, and A. Keromytis. Practical software diversification using in-place code randomization. In S. Jajodia, A. K. Ghosh, V. Subrahmanian, V. Swarup, C. Wang, and X. S. Wang, editors, *Moving Target Defense II*, volume 100 of *Advances in Information Security*, pages 175–202. Springer, 2013.

[15] S. Rauti and V. Leppänen. A survey on fake entities as a method to detect and monitor malicious activity. In *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 386–390, 2017.

[16] S. Rauti, J. Teuhola, and V. Leppänen. Diversifying SQL to Prevent Injection Attacks. In *Proceedings of Trustcom/BigDataSE/ISPA*, pages 344–351, 2015.

[17] S. Rauti, S. Lauren, J. Uitto, S. Hosseinzadeh, J. Ruohonen, S. Hyrynsalmi, and V. Leppänen. *A Survey on Internal Interfaces Used by Exploits and Implications on Interface Diversification*, pages 152–168. Springer International Publishing, 2016.

[18] D. Stanley, D. X., and E. Spafford. Improved kernel security through memory layout randomization. In *Performance Computing and Communications Conference (IPCCC), 2013 IEEE 32nd International*, pages 1–10, 2013.

[19] G. Tandon and P. Chan. Learning rules from system call arguments and sequences for anomaly detection. In *ICDM Workshop on Data Mining for Computer Security (DMSEC)*, pages 20–29, 2003.

[20] J. Uitto, S. Rauti, J.-M. Mäkelä, and V. Leppänen. Preventing Malicious Attacks by Diversifying Linux Shell Commands. In *Proceedings of the 14th Symposium on Programming Languages and Software Tools (SPLST'15)*, CEUR Workshop Proceedings 1525, 2015.