2014/2500M-0010

ISSN 1797-3457 (verkkojulkaisu)
ISBN (PDF) 978-951-25-2635-2

TIIVISTELMÄRAPORTTI (SUMMARY REPORT)

Wide application security by low-level program code obfuscation techniques

Ville Leppänen, Sampsa Rauti and Samuel Lauren
University of Turku, Department of Information Technology
ville.leppanen@utu.fi, sampsa.rauti@utu.fi

Abstract
The goal of our research project is to protect security of applications and software systems in a whole new way: by diversifying implementations of all the software layers and their interfaces on the binary level. The system call interface of the operating system is diversified uniquely for each system and all the entry points to this interface are diversified in applications and libraries accordingly. Moreover, the diversification in the library level is transitive. Malware that uses prior knowledge about existing interfaces in an operating system is now rendered useless because of diversification. All in all, our diversification based solution is a proactive solution against the prevailing operating system monoculture. The research project provided a proof-of-concept implementation for Linux. Our solution does not aim at removing the possible existence of security holes in Linux but rather making it infeasible to deliver effective malware through such security holes into the operating system environment.

1. Introduction

Malware, or malicious software, is one of the main problems in Internet today. McAfee Labs reported already in 2012 that malware sample discovery rate has accelerated to nearly 100,000 samples per day. Traditional fingerprint-based antivirus software is becoming increasingly inefficient in the fight against malware. Dynamic and behavioral malware prevention methods are reactive in the sense they only take action after malware has performed its malicious activities, and thus new kinds of approaches for security are needed.

Malware aims to misuse resources that are provided by execution platforms (operating systems) via access points implementing often very widely and publicly know interfaces. Malicious software uses prior knowledge about the identical interfaces of operating systems to accomplish its goals. To access resources on a computer, a malicious program has to know the interface that provides the resources. Because of the current operating system monoculture, an adversary can create a single malicious program that works on hundreds of millions of computers that use the same operating system.

Our goal is to provide protection to applications and software systems in a new way: We diversify the implementations of all software layers and their interfaces on the binary level. The system call interface of the operating system that can be used to access resources is diversified uniquely for each system and all the entry points to this interface are diversified accordingly.

Malware that uses prior knowledge about existing interfaces in an operating system is now rendered useless because of diversification. It can no longer use the resources of the system. Only trusted applications that are diversified will work in the system. Also, even if the malware were to find out the secret diversification for one system, it cannot perform any large-scale attacks because the diversification is unique for each system.

Therefore, our goal is to make the interfaces provided by the systems unique for each computer. This is made by diversifying all the software layers beginning from the kernel layer. When the software layers of the system have been obfuscated, the user applications used in the system also have to be diversified accordingly so that they will work in the system. The user of the system does not have to learn anything new and he or she will not notice any changes in the functionality of programs.

As the number of malicious programs keeps growing and new variants keep popping up, traditional fingerprint-based antivirus software is becoming increasingly inefficient in the fight against malicious programs. Also, antivirus programs often only detect the threats they are already aware of. Therefore, new approaches to malware prevention are needed to complement them. Our approach adopts a proactive view by preventing malicious code from harmfully interacting with its environment even before it is executed.

2. Research objectives and accomplishment plan

Our objective for the year 2014 was to create a software framework that can be used to diversify all the software layers starting from the kernel by applying a specific transformation method. Also, the goal was to successfully apply the transformation to user applications.

More precisely the research objective was to produce proof-of-concept solutions on the software framework for Linux operating system. As the core need is to diversify the system call interface of the operating system, the diversified interface also needs to be propagated to all the legal software instances within the diversified (operating) system instance. Those activities were modeled as the Phase I.

Most of the system calls are used from software libraries – e.g. from the libc library meant originally as a higher level interface of the system for C-language programs. Consequently, the same services coded by system calls are provided by such higher level library services to ordinary application, and applications are linked with respect to such libraries. While diversifying system call interface makes it difficult for malware to use the diversified system call interface, the necessary propagation of diversified system call interface to all legal “places” in the operating system environment make is again rather easy for malware to link to itself the diversified functionality from the standard libraries.

Therefore the necessary Phase II of the research was to diversify the software library layers in the same way. In practice, we needed to calculate so-called transitive closure of the library layer based on the usage of system calls and diversify them. Here the activity is similar as in Phase I: Access points (functions) need to diversified and respectively the changes need to propagated to all legal “places” within the execution environment (applications and libraries). Figure 1 illustrates the situation related to Phases I and II.

In practice the Phases I and II meant that ELF files of Linux operating system versions need to be rewritten, if the diversification is done only on binary code level. Doing diversification on source code level would be easier but more limited as source code on all applications might not be available. Therefore, the plan was to only work on binary files, the ELF files.

As a separate issue, the diversification of system call interface layer requires that the diversification is also implemented for the kernel. Easiest way to implement it on the kernel side is to modify the kernel source code and re-compile the kernel. That was chosen as the primary method, although we also studied (and mostly implemented) module based modification of Linux kernel (not requiring re-compiling).

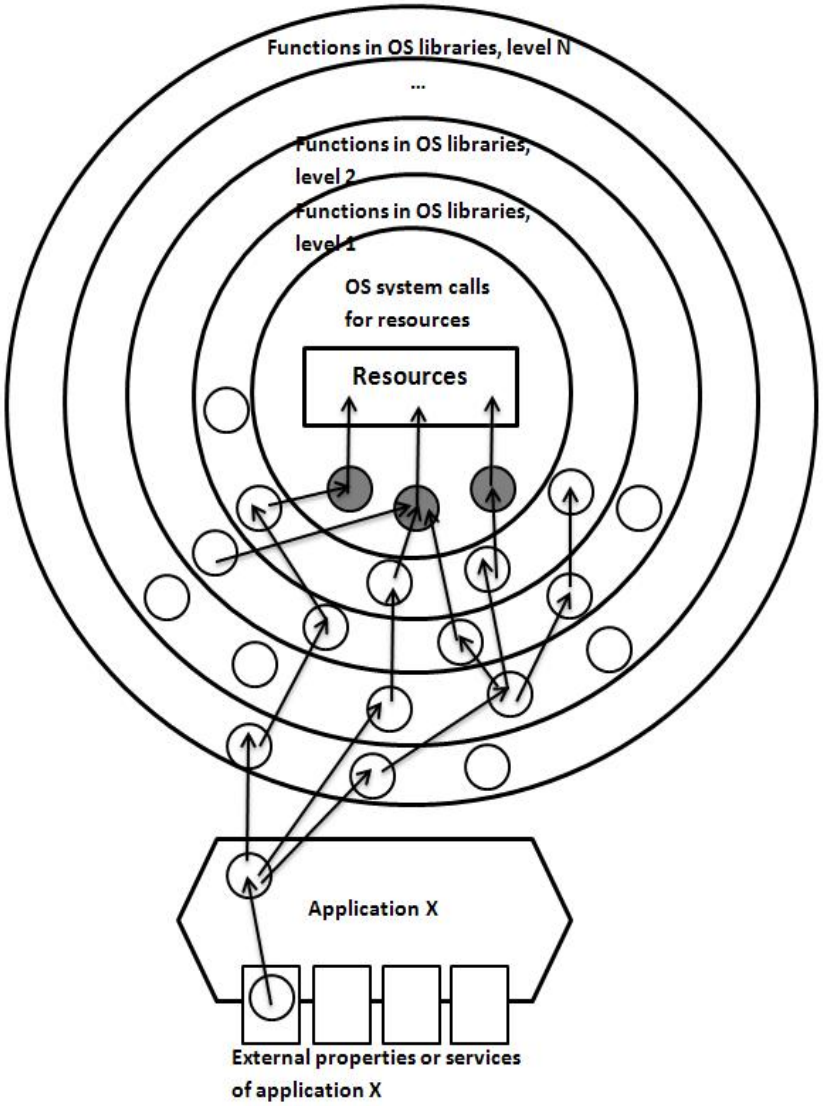

Figure 1: Operating diversification overview.

3. Materials and methods

Our methods consist of the conceptual operating system diversification scheme and its practical proof-of-concept implementations. Both will be described in what follows.

In our conceptual framework, we note that system calls are a fundamental set of services in an operating system. In order to interact with their environment, user applications use the services provided by the system call interface in a shared manner.

An application can invoke a system call using two different methods; either by calling the system call interface directly or using some library that provides wrapper for this specific system call. Thus, in order to prevent a piece of malware from accessing the system call interface we have to block both direct and indirect system calls originating from untrusted applications.

Here, the term diversification refers to meaning-preserving mapping in a programming language. That

is, the program's semantics are preserved even though the code is transformed to a different form. Therefore, the user of the program does not experience visible change when using the program. In this paper, diversification simply refers to renaming symbols in Linux binaries (the second part of our scheme mentioned above).

The first part of our diversification scheme, system call diversification, means that the system call numbers defined in the operating system kernel are changed. All the program code in libraries and user applications that invokes system calls directly using these numbers must therefore be diversified accordingly as well, or it will not work anymore.

The second part of the diversification deals with the cases where a system call is not invoked directly. Instead, the call travels through several software layers before reaching the system call implementation. A system call issued by a user application usually moves through several layers or libraries in the operating system. If we are to prevent an untrusted application from making any system calls we must hide all possible entry points it could use to gain access to system calls. This is why changes in function signatures must be transitively propagated to all trusted operating system library layers and finally to the user applications.

These trusted layers and applications are diversified with a secret diversifying function which makes them compatible with all the inner layers. The end result is that the critical entry points of the whole system are diversified, rendering untrusted applications and malware useless.

The functions directly or indirectly calling the system call implementations forms the transitive closure. These functions are all diversified, which can be implemented for example by renaming their symbols in the binary files, like we have done in our project. This will effectively prevent a malicious program from using these entry points to access a computer's resources because the malware authors do not know the diversification secret (the function defining the relation between original and new symbol names). On the other hand, applications that are diversified correctly can still use the services provided by system calls.

The trusted software layers and user applications are diversified with a secret diversification function which makes them compatible with new system call numbers defined in the kernel. As a result, the entry points that directly lead to the system calls are diversified in the whole system, preventing malware and any untrusted applications from using them to invoke system calls. By diversifying the library entry points leading to system calls, we are preventing the scenario in which a malicious program loads the library and utilizes them to achieve its goals.

Our scheme is meant to protect computers from the malicious code that is either executed as its own process or as a part of another executing process. In our scheme, the assumption is that a malicious program has no access to the file system, for example, and cannot analyze files or our secret diversification function that has been applied to binary files. The reason for this assumption is that accessing the file system requires invoking system calls and thus an external malware does not have an easy way to access the file system.

Together with the system call number diversification, diversifying the transitive closure makes it much more difficult for malicious programs to use any services on a computer. Untrusted programs know neither the names of functions nor the system call numbers in other applications or libraries that lead to system calls.

Corresponding the two parts of our scheme, the diversification of both direct and indirect system calls, we have built two tools for these purposes. Additionally, we have also conducted a study that estimates the extent of diversification work needed to diversify a complete Linux operating system distribution.

The first tool concentrates on diversifying all the direct system calls found in the binaries of a system. This proof-of-concept tool rewrites the system calls in x86-64 ELF-64 binaries by making use of a simple linear sweep algorithm. This is a straightforward disassembly method that decodes everything appearing in sections of the executable that are typically reserved for machine code. We limit the analysis to executable PROGBITS-type sections in ELF binaries. The diversification is done after compile time before the executable file is run.

The tool tries to find system calls by walking through the program code sections from the beginning to the end. It looks for SYSCALL commands used in x86-64 architecture. When such a command is found, it starts searching the system call number associated with this call. This is done by backtracking from the location of SYSCALL command and trying to find the command where the system call number is set. As the number of system call to be invoked is put into a register, our tool looks for commands that change values of specific registers related to invoking system calls.

The second tool we implemented diversifies all the indirect calls to the system call entry points found in a binary file and the libraries it depends on. The tool is based on the fact that libraries and programs participating in the dynamic linking contain symbolic names for the functions, variables and other data they either provide for others to use or expect to find from external ELF binaries.

The tool we have implemented diversifies, or renames, the symbols in shared libraries in Linux operating system. Furthermore, we transitively propagate these changes to trusted ELF binaries which depend on the entities referenced by those symbols. The relation between the original and diversified symbol names is kept secret. This should make it very difficult for an adversary to write functioning ELF binaries for the system, as the names for needed symbols in the ELF binary symbol table must match the names in the binary that provides these symbols.

The core of our symbol diversification tool consists of three separate tools. Each of these handles one step of the overall diversification process. The first tool, symbol collector, handles the collection of symbol from 64-bit ELF binaries and outputs a simple plain text listing of them. The second step, symbol transformer, takes the previously produced symbol listing and transforms each symbol using some predefined method. The last step of our process, symbol rewriter, takes the original binary and the now transformed symbol definitions and produces a modified version of the binary using this information. By clearly separating these steps we hoped to maximize the flexibility of our diversification pipeline and allow for future enhancements.

4. Results and discussion

Our main contributions are a scheme for large-scale system call diversification for operating system protection and implementations for concrete diversifier tools to demonstrate feasibility of our approach. In particular, we developed following kind of software:

- For modifying system calls, our implementation consists of approx 2,300 lines of (C, C++) code.
- For modifying interfaces of transitive closure (of system calls) in ELF files, our implementation consists of 2,900 lines of code (C++, scripts).

For the tool diversifying the direct system calls in binary files, our experiments show that a large majority of system calls is handled well by our tool, but there are still some challenges. However, the numbers of unidentified calls were usually relatively small for analyzed binary files. For example, in Gentoo distribution, out tool successfully diversified about 92 % of all system calls. Our tool has also been successfully tested on Linux Fedora distribution.

To overcome the challenges, we have also studied several ways to increase the accuracy of diversifying the direct system calls in binaries to 100 %. The small total amount of system calls also makes things easier. As we have seen, very few binaries in the tested Linux distributions contained direct system calls. Most direct system calls are in standard libraries and well-known command line tools, not in the normal user applications.

We also run several tests with our other tool, the symbol diversifier. In our experiments, it was found that the binary itself and all the libraries dependent of the desktop programs were successfully diversified by our diversification tool. Many of these libraries were im-

portant graphical libraries that are shared by many desktop applications.

As an example, we can were able to diversify the whole CRUX 3.0 Linux (1.1 GB; 70,632 files but only 3,028 ELFs (libraries and applications)) in about 10 minutes.

A few minor issues were discovered. There were some problems with libraries loaded dynamically at run-time. For example, the dynamically loaded plugin may try to use a symbol provided by libc, but this symbol is not found in the new diversified version. However, we have presented solutions to overcome this issue. All the minor errors found are well understood and did not have any critical effects on the functionality of the program.

Based on these results, we believe system call diversification is a feasible approach for protecting operating systems from malware. This is especially true for systems where a certain set of well-known libraries and applications is used and in many embedded systems that are more restricted in nature.

We also studied diversifying the kernel implementation. The result of kernel analysis is that of 151,845 functions only 23,835 use system calls, but it is still wise to diversify all interfaces (as there are only 600,000+ places in kernel where modifications need to be made).

5. Conclusions

We have created a scheme for diversifying all system calls, library entry points and user applications in a system. In our approach, system call entry points and any direct system calls in the binary files are diversified.

We have also built several concrete diversification tools for this purpose. These tools and their components have been discussed in detail in our publications. We have also provided concrete evidence on the feasibility of our approach through diversification experiments with popular desktop applications and the libraries they depend on. Approaches to overcome challenges present in the diversification process have been discussed.

Despite some challenges, our tools have been able to successfully diversify executables so that the functionality of the system is not affected. Our experiments show that system call and API diversification are feasible approaches to protect applications and systems from attacks.

During this project there have been outside interest towards our research efforts. However, as the implementations are far from being products. Moreover, even the current tool would require a lot more testing as there probably exists Linux configurations for which our implementation is not fully feasible.

The major future works are related e.g.

- to wider practical experimentation of the current implementation,
- to adaptation to other Linux-like operating systems,
- to adapting the implementation to Microsoft operating systems (there the file format is similar to ELF but different and also the system call interface is different),
- to strengthening the overall implementation by also applying idempotent transformations (called obfuscations) within the code to hide its meaning from potential malicious parties – there the idea is to try to prevent reverse-engineering of propagation places of diversified implementation so that the malicious party will not be able to find out the secret diversification function based on code samples,
- to applying the developed technique to industrial embedded systems,
- to applying the diversification to other kind (higher level) interfaces,
- to integration of developed software tools,
- to considering diversification and software updates – how updates can be easily applied to

diversified applications (how to diversify the update packages before applying them, and

- to studying diversification secret (key) management issues.

6. Scientific publishing and other reports produced by the research project

1. S. Rauti, S. Lauren, S. Hosseinzadeh, J-M. Mäkelä, S. Hyrynsalmi, V. Leppänen: “Diversification of system calls in Linux binaries”, Proceedings of the 6th International conference on Trustworthy Systems (InTrust 2014), December, Beijing, China; 15 pages, to appear in the Lecture Notes in Computer Science series of Springer (April 2015).

In this paper, we analyze the presence of system calls in the ELF binaries. We study the locations of system calls in the software layers of Linux and also examine how many binaries in the whole system use system calls. Additionally, we discuss the different ways system calls are coded in ELF binaries and the challenges this causes for the diversification process. Also, we present a diversification tool and also suggest several solutions to overcome the difficulties faced in system call diversification. The amount of problematic system calls is small, and our diversification tool manages to diversify the clear majority of system calls present in standard-like Linux configurations. For diversifying all the remaining system calls, we consider several possible approaches.

2. S. Lauren, P. Mäki, S. Rauti, S. Hosseinzadeh, S. Hyrynsalmi, V. Leppänen: “Symbol diversification of Linux binaries”, Proceedings of World Congress on Internet Security (WorldCIS-2014), 6 pages, to appear, IEEE, London, UK, December 2014.

In this paper, the idea is to diversify all the indirect library entry points to the system calls on a specific computer. As a result, it becomes very difficult for a piece of malware to access resources. The diversification of indirect system call entry points in operating system libraries is unique for each computer. Therefore, a piece of malware no longer works on several computers and becomes incompatible with their environment. We also present a concrete diversification tool and results on successful diversification. We conclude that despite some challenges, our tool can successfully diversify symbols in binaries and associated libraries in order to protect the system from attacks.

3. S. Rauti, J. Holvitie, V. Leppänen: ”Towards a diversification framework for operating system protection”, In proceedings of International Conference on Computer Systems and Technologies (CompSysTech), ACM Press, ACM ICPS 883, pages 286-293, 2014.

In this paper, we discuss a conceptual framework for operating system protection by diversification. In addition, as our main contribution we also examine the extent of work needed to diversify the Linux kernel so that the interfaces providing the critical resources are different for each system. Our work lays the basis for a practical implementation of an operating system diversifier.