2015 IEEE/ACM 1st International Workshop on Software Protection

Obfuscating Windows DLLs

Bert Abrath, Bart Coppens, Stijn Volckaert, and Bjorn De Sutter
Computer Systems Lab
Department of Electronics and Information Systems
Ghent University, Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium
E-mail: {bert.abrath, bart.coppens, stijn.volckaert, bjorn.desutter}@elis.ugent.be.

***Abstract*—We present two techniques to obfuscate the interfaces between application binaries and Windows system DLLs (dynamic-link libraries). The first technique obfuscates the related symbol information in the binary to prevent static analyses from identifying the invoked library functions. The second technique combines static linking with code obfuscation to avoid the external interface altogether, thus preventing dynamic attacks as well. This is done while still maintaining compatibility with multiple Windows versions, through run-time adaptation of the application. As the first concrete result of this ongoing research, we demonstrate and evaluate the techniques using a proof-of-concept tool applied to a simple test program.**

I. Introduction

To prevent attackers from easily reverse engineering the distributed binaries, a software vendor can obfuscate the binaries he distributes, i.e., transform them into semantically equivalent ones that are significantly harder to understand and reverse engineer. However, when a vendor deploys an obfuscator tool on his software, this tool will only transform the code available at obfuscation time. Shared libraries of third parties over which the vendor has no control will not be obfuscated. This leaves the external interfaces of his software to these third-party libraries unprotected, which makes them an ideal starting point for static reverse engineering attacks. Furthermore, the interface between an application and the external libraries can be used by attackers to observe and modify the program behavior dynamically.

This large attack surface – caused by dynamically linking a program against shared libraries – can be removed by statically linking the libraries into the program and applying the obfuscations to the whole resulting binary. Besides obfuscating the linked-in application code and the linked-in library code, this also enables obfuscation of the interface between the application and library code.

On GNU/Linux, static linking is straightforward. Statically linked versions of the standard C libraries exist, their source code is available for obfuscation, and the interface between those libraries and the kernel remains stable. This suffices to ensure that applications statically linked against old C libraries can still interface with newer versions of the kernel. The interface between the library and the kernel is still available to the attacker as a starting point for static or dynamic attacks, but at least we can ensure that static analysis can no longer easily determine which program code interfaces with the kernel.

On Windows, however, there is no support for statically linking complete binaries. First, the core Windows API is only available in the form of a set of dynamic-link libraries (DLLs), such as user32, kernel32, etc. Of these system libraries no versions exist that are statically linkable into programs. Secondly, the interface between these system DLLs and the Windows kernel varies between different Windows versions. For example, one cannot take ntdll, which is a core DLL, from a Windows 8 installation and use it on a Windows 8.1 system: the system call numbers in both versions simply do not match.

In this paper, we propose two automated techniques that mitigate DLL-based attacks on Windows binaries. The first technique acts on the meta-information stored in the binaries' import tables. This information informs the Windows dynamic loader about which functions are to be imported from which libraries. We remove this meta-information from the binary and replace it with a custom loader in the binary that loads the required libraries and imports the required symbols. This prevents static analyses from identifying the external library functions invoked in the binary. However, the meta-information is still present in some form and the libraries are still linked dynamically, which leaves the program vulnerable to dynamic analyses and run-time tampering with its behavior.

We therefore also propose to statically link the Windows system DLLs into the binary and to obfuscate the resulting binary, including the linked-in DLLs. Our experiments show that this technique can to a large extent deal with the currently observed variations between different Windows kernel interfaces. To support this, we extended the Diablo link-time rewriting framework [1], which can now link application code with code that is automatically extracted from Windows DLLs. This code is augmented with glue code to allow the program to adapt itself to the system call interface in use on the platform on which the binary is executed. The resulting statically linked program hence no longer executes any code loaded from dynamically linked system libraries, and is ready for combined obfuscation of the application code and the library code.

As a proof-of-concept, we succesfully applied different obfuscation techniques to the Windows API functionality linked into a simple test program that calls functionality of user32 and kernel32.

The rest of this paper is structured as follows: Section II describes the related work, which is followed with a description of how Windows system DLLs function in Section III. Section IV describes the technique for meta-information encryption, and Section V describes our technique of statically linking Windows DLLs. Section VI details some of the chal-

978-1-4673-7094-3/15 $31.00 © 2015 IEEE
DOI 10.1109/SPRO.2015.13

lenges we faced to automatically transform Windows binaries. This is followed by a discussion of the current limitations of our techniques in Section VII. Section VIII describes our experimental evaluation of our technique. Finally, we present our conclusions in Section IX.

II. Related Work

Reverse engineers and hackers have many techniques to attack and understand binaries. One set of related attack techniques is based on the fact that most programs depend on external library code being loaded and executed as part of the program. Because the links between the application and its libraries are only resolved at load time and because they are stored in the form of easily readable symbol information, attackers can study which external functionality the program requires and influence the linking process to their own advantage. In particular, attackers can *hook* a shared library, which means that run-time invocations to shared library functionality are redirected to code controlled by the attacker. This interception of library calls allows attackers to observe the dynamic behavior of a program by monitoring the library calls and the passed data (a.k.a sniffing). It also allows them to modify the data passed from the binary to library and back (a.k.a. spoofing), to skip the library calls, and to check or intervene in the program state in any other useful way.

More concretely, attackers use PE header editors such as LordPE [2] to change imported libraries and symbols manually and to replace functionality using the Detours framework [3] either by forcing a dependency on additional DLLs, or by injecting such code at run time [4]. One of the more popular targets of such attacks are games [5].

Static attacks can benefit from the symbol information present to improve the analysis of the program code that calls into the libraries: knowledge of API calls in the program can provide valuable information about the usage of local variables and arguments without having to observe the program at run time. For example, OllyDbg [6] and IDA [7] – two tools often used in the reverse engineering of Windows binaries – both recognize calls to the Windows API and annotate these using the names and types of the arguments, improving understanding of the program.

To defend against the aforementioned attacks, static linking can be used. For Windows, however, such static linking is typically limited to libraries from one vendor or from multiple cooperating vendors. On Windows, static linking does not include the system libraries.

On Linux, static linking can include the standard system libraries. It has been shown, however, that merely using static linking and then stripping all symbol information from the binary is not sufficient to prevent the static identification of library functions by the attacker.

In fact, multiple techniques exist to identify library functions in binaries. IDA Pro uses a technique called FLIRT, which uses pattern matching to detect and identify the presence of compiler-generated patterns and certain library functions, and can be extended with user-provided databases [8]. On Linux, system library functions can be identified by the system call numbers they embed to interface with the kernel, and then one can iteratively try to identify the callers of the already identified functions [9]. Machine learning can be used to automatically learn patterns required to identify library functions [10]. Finally, a more generic technique to identify statically linked library code is implemented in BinDiff [11]. This diffing tool uses the structure of control flow graphs, call graphs, basic blocks, etc., to identify matching functions between two binaries [12], [13]. Such a tool can also be used to match the functions in a separate library with those in an application binary into which that library has (potentially) been linked, thus identifying those functions in the binary.

Obfuscators are used to prevent reverse engineers from easily analysing binaries and from determining the functionality of code. Obfuscators have as input (part of) a program and transform this to thwart analyses and comprehension [14]. Obfuscating transformations can also be applied to thwart diffing tools: by randomizing the concrete application of many obfuscating (and optimizing) transformations on a piece of code many different versions of an application or library can be generated in which tools like BinDiff can no longer identify the corresponding fragments [15], [16].

Obfuscators can act on different levels of program abstraction. First, they can rewrite program source code. For example, the Tigress diversifying compiler [17] and the Cloakware diversification system [18] are obfuscators that apply source-to-source transformations on C code. At a lower level of abstraction are obfuscators that transform the compiler representation of code, such as obfuscator-llvm [19]. At the lowest level are obfuscators that rewrite a program's binary instructions, such as is the case with the obfuscator front-end of the Diablo link-time rewriter [20].

For the techniques presented in this paper, we can only rewrite binary code because no source code of the Windows system libraries is available to third-party software developers. Several frameworks for binary rewriting other than Diablo exist. McSema [21] has been demonstrated to to rewrite Windows DLLs such as kernel32. Other such tools that can rewrite or decompile native code into a functional intermediate representation include SecondWrite [22], Dagger [23] and Fracture [24], which translate binary code into LLVM IR. In particular, SecondWrite has been used to rewrite statically linked binaries without symbol information [25].

Furthermore, there exist approaches to bundle DLLs with applications. Tool such as PEBundle [26], MoleBox [27], and DLLPackager [28] bundle DLLs with the application, and add code to unpack these DLLs and resolve the required symbols. However, such applications are still being linked dynamically.

Finally, it is worth mentioning that some forms of importless binary Windows code are already used in practice. In particular, shellcode that is injected into a process can usually not rely on the program loader to load libraries and resolve symbols. Thus, some shellcode already uses techniques similar to ours to locate symbols in kernel32 at run time [29]. Similarly, these techniques have been extended to craft proof-

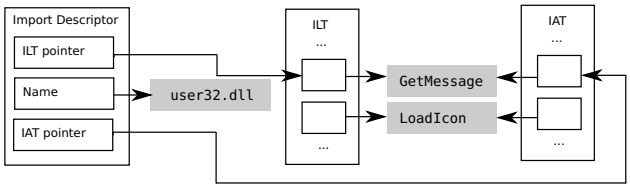


Fig. 1: An example of the import tables of a PE file.

of-concept importless PE binaries [30]. However, none of these existing techniques are automated.

III. Background: System Libraries on Windows

To describe the context in which our techniques are applied, we first describe how applications import symbols from libraries on Windows and how these libraries are loaded at run time. Next, we discuss how the system libraries interface with the Windows kernel.

A. Library Loading

To understand some of the issues with statically linking applications with Windows DLLs, we discuss in some detail how DLLs are structured. On Windows, both DLLs and executable binaries are stored in *Portable Executable* (PE) files. Dynamic linking of such files is allowed through the meta-information stored in the PE file headers. This meta-information allows files to declare the symbols they require from external sources in import tables, and to declare the symbols they expose in export tables.

Both executables and DLLs can import and export symbols. Since this paper aims to hide the imported symbols, we discuss the importing of symbols in more detail. Figure 1 shows the import tables of an application that imports the GetMessage and LoadIcon symbols from the user32 library. There is an *Import Descriptor* for each library from which symbols are imported. This descriptor contains references to the name of the library, to an offset in the *Import Lookup Table* (ILT), and to an offset in the *Import Address Table* (IAT). The ILT contains references to the names of symbols that need to be imported. These symbols are searched for in the *Export Name Table* (ENT) of the corresponding library.

When the Windows loader loads a PE file it iterates over the import descriptors, loading the libraries the file depends on. The loader iterates over the ILT entries, calculates the addresses at which the corresponding symbol have been loaded, and writes these in the corresponding IAT entries.

Applications can thus refer to imported symbols indirectly through the IAT: they refer directly to the address of the IAT entry, which then contains the address of the actual symbol. For example, in the binary whose import tables are shown on Figure 1, a call to the LoadIcon function would be encoded as call [IAT+offset1], where offset1 refers to the offset of the IAT entry that corresponds to LoadIcon. Instructions such as these call instructions refer to the absolute address at which the compiler expects to find the IAT. To support libraries being loaded at different addresses – for example to support Address Space Layout Randomization – the loader rewrites all instructions that refer to the IAT such that they use the actual load address. This is achieved by means of so-called base-relocations that are stored in a PE-file's .reloc section.

B. Windows System Calls

When a Windows binary performs high-level API calls – i.e., invokes system library functionality – the control flow will almost always be redirected into the kernel at some point. However, the high-level API functions invoked by the program will typically not interact with the kernel itself. Instead, they will set the stage for a call to a small, low-level function in the core Windows DLLs, which in turn calls into one of the kernel's system services. These low-level functions are *System Call Wrappers* (SCWs). They only set the correct arguments for the system services, after which they perform the system call. The Windows kernel knows how to direct the execution to the correct system service by means of the *System Call Number* (SCN) set by the SCW.

As an example, take the high-level action of creating a new file with the CreateFileW function. On Windows 8, this code resides in KernelBase. This DLL contains the high-level implementation required, but ultimately calls NtCreateFile in ntdll. This low-level function then sets the SCN value to 84, which corresponds to the NtCreateFile system service on this system, and performs the actual system call.

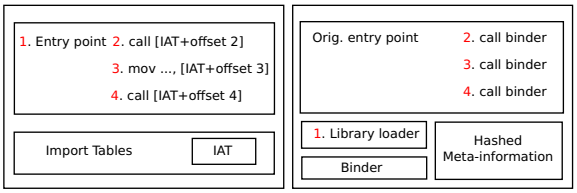
The mapping between the SCNs and the system services differs from one Windows version to the other. Reverse engineered tables of these mappings are available that list all SCNs for different versions of Windows [31].

IV. Removing Program Import Information

Our first technique attempts to hide the interface between the protected application and its libraries by completely removing the IAT from the application binary. Without the IAT, static analysis tools can no longer determine the external functions and libraries imported by the application and thus provide less useful information to reverse engineers. Similarly, since the IAT is removed from the application, IAT hooking attacks are no longer possible either.

To keep the program functionality intact after removing the IAT, our link-time rewriter built on top of Diablo (http://diablo.elis.ugent.be) rewrites the program such that it emulates the Windows dynamic linking process itself. In particular, our tool injects a custom loader into the program and rewrites the code fragments that access the IAT in the original program. The rewritten program operates as follows:

- The custom loader loads the DLLs into memory on program initialization. Rather than using library names from a program header, it uses *hashed* library names. The used hash function can be diversified – i.e., customized – in each instance of the program to prevent class attacks on this custom loader.
- In the original program code, each instruction that indirectly invokes library code or indirectly accesses library



(a) Original program (b) Rewritten program

Fig. 2: The program's static meta-information and structure, for both the original and the protected application.

data through the IAT is transformed into a call instruction that invokes the newly inserted *binder* function.

- When the binder is invoked, it performs a lookup in an injected custom hashed table to identify the intended target function or data and rewrites the call instruction into an instruction that performs the same functionality as the original instruction, but now does so directly, i.e., without going through the IAT.

Figure 2 shows a schematic representation of a program before and after applying our protection. We next discuss in more detail the custom library loader and how we transform IAT-dependent instructions.

A. Custom Library Loader

The original program entry point (point 1 in Figure 2 (a)) is replaced with a newly inserted library loader function (point 1 in Figure 2 (b)) that replaces the operating system's dynamic linker. It first loads the dynamic libraries on which the program depends, but does not yet calculate the addresses of the symbols imported by the program; this is defered to when the IAT-dependent instructions are actually executed.

Our tool inserts a custom table in the program that contains hashed library names. The loader iterates over all these library names and matches them with the hashed names of the libraries located on the user's system. When a matching hashed library name is found, the corresponding library is loaded into the program memory and the necessary bookkeeping operations are performed (PE Files also support *ordinal-based* symbol lookup; our prototype implementation does not yet support this).

To load dynamic libraries into program memory, we make use of the LoadLibrary function provided by the Windows API. This function is located in the kernel32 library which is *always* loaded into the program memory by the kernel, even if a binary does not import anything at all. Our loader uses a Windows data structure called the *Process Environment Block* (PEB) to find the load address of kernel32.

B. Transforming IAT-dependent Instructions

In the original binaries all application code that accesses imported symbols does so through an indirection through the IAT. See points 2, 3, and 4 in Figure 2 (a). Our tool identifies all those references and replaces the original instructions with calls to the binder code, as shown in Figure 2 (b). The call instruction pushes a return address onto the stack, which the binder can use to index a lookup table.

The elements of the lookup table contain all possible continuation points from which the binder can be called. To prevent simple lookup attacks, the table contains hashed (and salted) return addresses. For each call site, the table contains:

- *A reference to the custom DLL table*, to inform the binder in which of the DLLs the symbol should be resolved.
- *A hashed symbol name*. The binder iterates over all symbols exported by the referenced DLL to find a match for this hashed symbol name.
- *Information on the original instruction*. Because the original instruction referencing the IAT was not necessarily a call instruction, we also store the original instruction opcodes in the table. For example, in Figure 2 (a), we have instructions referencing to the imported symbols as functions as well as instructions that refer to the imported symbols as a variable. Furthermore, the table also stores information on the offset in the instruction where the IAT-symbol is referenced.

When in the example in Figure 2 (b) a call to the binder executes, the binder thus finds the relevant table entry, and resolves the imported symbol. Next, the binder (using VirtualProtect calls) remaps the memory containing the call site as writeable, and replaces the call to binder with the appropriate original instruction, such as call or mov in the example in Figure 2 (a). The binder then updates this instruction with the resolved symbol address, and remaps the code as executable. The binder changes the return address stored on the stack into that of the rewritten instruction and returns, so that the original instruction is executed. The next time this instruction is reached, it will be executed as is, i.e., without requiring another intervention from the binder.

V. Static Linking of Windows Binaries

Even after removing the IAT, dynamic attacks are still possible by modifying the DLLs from which symbols are imported. To solve this, we propose to statically link all required DLLs into the binary and then possibly obfuscate the resulting binary.

To statically link a Windows binary we first compute the transitive set of library dependencies. Our prototype then links those libraries into the application, after which instructions that indirectly refer to library symbols through the IAT are rewritten to refer directly to the now linked-in symbols instead.

Each DLL can contain an initialization function that is called by the program loader once it has loaded the DLL and it has resolved all dependencies. However, in a statically linked program, the program loader no longer performs any actions on behalf of the code that has been statically linked into the program. Thus, our tool inserts initialization code into the rewritten binary that calls the appropriate initialization routines on program startup in the correct order.

As discussed earlier, the interface offered by the Windows kernel differs between Windows versions. The initial statically linked binary thus cannot be run on Windows versions other

than that from which the linked-in system DLLs originate. To make the statically linked binary compatible with multiple Windows versions, the binary has to adapt dynamically to the version of Windows on which it is running.

To that extent, our tool injects additional functionality into the program to update the encoded SCNs similarly to how we update references to IAT-dependent instructions as discussed in Section IV. We replace all SCWs with a code fragment that loads a hash of the name of the SCW, followed by a call to another binder function that will set the correct SCN.

To determine the correct SCN we start with locating the SCW, which is exported as a symbol by ntdll. Because ntdll is loaded in any process – even those without an import table – the binder can scan over all its exported symbols and find a match for the hash. SCWs have a simple structure and so we can easily extract the SCN from the corresponding SCW in the loaded ntdll, which by definition contains the correct SCNs.

The resulting statically linked binaries can then be transformed and obfuscated further by Diablo.

VI. Rewriting PE Files

All techniques in this paper build on the capabilities of the Diablo link-time rewriter, which is able to re-link binaries and to apply transformations on the code being re-linked. To disassemble the code correctly and to build correct intermediate representations of the code to be transformed, Diablo relies extensively on symbol and relocation information available in the re-linked object files. Some compiler tool chains (such as ARM's proprietary compilers) already provide all needed information in the object files they generate. For other tools like GCC, LLVM, and binutils, a set of patches is available to make them produce sufficient information such as non-relaxed relocations and so-called mapping symbols, e.g., to differentiate between code and data in code sections.

The object files in Windows system libraries do not contain sufficient information, however, and we can obviously not regenerate them with a patched Visual Studio tool chain.

To enable Diablo to disassemble Windows binaries and reconstruct their control flow accurately enough, its linear sweep disassembler (which originally started linearly disassembling all instructions at all mapping symbols indicating the start of a code fragment) was extended into a recursive descent disassembler. This built-in disassembler could also be complemented with more advanced, interactive disassemblers such as IDA Pro's to make sure that all code in the libraries is effectively treated correctly as code. This is similar to the approach that is used by McSema [21]. Since there are not many different versions of the system libraries, it suffices to identify the code once manually using IDA Pro, and to store the data – i.e., identified code ranges – in IDA Pro's database for later reuse in Diablo.

VII. Limitations

We now discuss some of the limitations of the proposed techniques.

A. Limitations of our Prototype Implementation

First, our current implementation does not support multithreaded applications yet. In order to make the temporary mapping of code pages as writable – rather than executable memory – function correctly in multithreaded applications, all binder operations need to be encapsulated in critical sections. Doing so correctly is simple, but doing so with as little overhead as possible requires more research.

Secondly, our current prototype implementation does not yet initialize all statically linked system libraries completely. A lot of engineering and reverse-engineering is needed to complete this, for which we lack the resources. There are no fundamental issues, however, and for people with more documentation about the internal operation of the libraries – such as Microsoft's developers – this would be much easier.

B. Copyright Issues

While this paper presents techniques to link Windows DLLs statically and obfuscate the resulting code, the Windows end-user license agreement currently does not allow a vendor to modify and then redistribute Microsoft code.

C. Security and Compatibility Issues

One of the advantages of dynamically linking against libraries (rather than statically) is that once a security issue is addressed in some library code, only a single instance of this code – i.e., the one installed version of the library – needs to be patched by the user on his system. Statically linking against a library means that each individual vendor is required to re-link (and re-transform) the binaries and to re-distribute these to their users, which then need to update all binaries. While this is clearly a sub-optimal scenario, statically linking binaries already happens in practice – for example in the Morpheus Linux distribution (http://morpheus.2f30.org/) – so apparently this issue does not outweigh the advantages of static linking in all scenarios. Most importantly, in this age of distributing software via the internet updating binaries has become trivial, unlike in the age where software was distributed via physical media such as CDs. Indeed, most software is currently downloaded, and all somewhat user-friendly web pages immediately present the correct version of software to be downloaded, such as the latest version for Windows, Linux, or Mac matching the user's OS, OS version, installed GPU, etc.

For the same reason, most issues related to compatibility across different versions of Windows could be handled easily these days. We briefly discuss four such issues.

First, the structure of SCWs can change between Windows versions. This is important because our binder determines the SCN by analysing the SCW. Our prototype currently supports the different structures of SCWs used by both Windows 7 and Windows 8/8.1. However, it is possible that in future versions of Windows the SCWs have a new structure. Of course no tool can completely foresee the changes in future versions. However, whenever a user installs a new version of Windows, he will likely download his applications again. So at that time,

he can download a version adapted to the new structures in his new Windows version.

Secondly, the system services can change between Windows versions as well, as they are considered an OS internal matter by Microsoft. For example, the implementation of WriteFile on Windows 7 internally redirects the control flow to either the WriteConsole function, or to the NtWriteFile SCW, depending on whether or not the target of the write is a console window. However, in Windows 8, the NtWriteFile SCW is always used. Our current implementation has special cases built in for all such function calls we encountered in our tests. Thus, we can guarantee backwards compatibility, but we cannot guarantee a future-proof behavior. Again, that is not an issue, as a user can install adapted versions of his application after he has upgraded his OS.

Thirdly, the details of the internal state kept by ntdll differ across versions. Thus, the state initialized by the ntdll automatically loaded by Windows can differ from how the ntdll statically linked into the program expects it. Again, our prototype has support for the cases we encountered by forcibly having the rewritten program re-initialize certain data structures on program startup.

Finally, our current implementation scans the exported symbols in ntdll to find the SCW symbols. However, it is not required that SCW symbols are actually exported. In particular, this is not the case for some existing GUI-related SCWs located in gdi32, which we thus do not support yet. Future versions of Windows could similarly no longer export the actual SCWs located in ntdll. To solve these issues, our tool could inject more complex pattern matching code into the binaries, with which all necessary code fragments can be identified for at least the OS versions that exist at a certain point in time. We could benefit from the Windows library OS implementation of project Drawbridge, which is a research project in which the interface between userspace and kernelspace is dramatically simplified [32], which could reduce the amount of pattern matching required by our tool.

VIII. Experimental evaluation

We evaluated our prototype implementation of the two presented techniques on a simple test program. It contains a couple of simple API library calls to write to files and to the standard output. This program was compiled as a 32-bit binary with Microsoft Visual Studio 2013 Ultimate. The resulting transformed binaries were tested on multiple 64-bit versions of Windows: Windows 7, Windows 8 and Windows 8.1.

First, we applied our IAT-removal technique. The transformed program still executes correctly on all our tested versions of Windows. We opened the binary in PEView (http://wjradburn.com/software/) and verified that indeed the IAT is no longer present in the resulting binary. Because of the inserted binder and loader code, and because our inserted tables have large records, there is a significant program size increase: from 2560 bytes for the original program, to 17408 for the protected program.

static linking	2385408
random layout	2477568
obfuscated	2607104

(a) Rewritten programs

original	2560
kernel32	1036288
ntdll	1467384
KernelBase	838144

(b) Input files

Fig. 3: File size information.

We evaluated the static linking and transforming of Windows DLLs with the same binary. We transformed the binary in three different ways:

1) *Statically linked binary*: We statically linked all dependencies into the transformed program and removed the code of which Diablo's analyses concluded that it was unreachable. The DLLs statically linked into the binary are taken from a Windows 8.1 installation.
2) *Random layout*: After statically linking the dependencies and removing unreachable code, Diablo's code layout engine is instructed to use a pseudo-random number generator to determine the placement of code fragments in the final binary.
3) *Obfuscated + Random Layout*: Before applying the randomized layout scheme of the previous step, we also obfuscate the code in Diablo's intermediate representation. The applied obfuscation technique consists of a simple scheme for inserting branch functions in basic blocks [33].

All the resulting statically linked binaries worked correctly on all our installed Windows versions, thus demonstrating the functioning of the different binder techniques.

Figure 3 shows how the file sizes are affected by the different transformations. As Figure 3 (a) shows, statically linking the DLLs into the binary significantly increases the file size. Even though Diablo removes unreachable code, in their present form the analyses are not precise enough to remove all unused code or data. Still, when we compare the resulting file size against the file sizes of the input binaries shown in Figure 3 (b), we observe that the final binary is still significantly smaller than the sum of the input file sizes. Finally, it is clear that the branch function obfuscation also increases the file size, albeit not dramatically in this experiment, due to the rather limited deployment of the technique.

We also evaluated the effectiveness of static linking against static attacks to identify library functions. Such attacks can currently not use techniques like FLIRT, because signature databases do not yet contain the Windows API libraries. In the arms race between attackers and defenders, as long as there is no static linking of Windows API libraries, no tools are developed to identify fragments in such statically linked libraries. For that reason, we cannot evaluate such tools yet.

Instead we evaluated the effectiveness of an attacker that tries to identify library functions in the statically linked binary by diffing that binary with the original library containing those functions. If the diffing tool finds a match, the attacker can

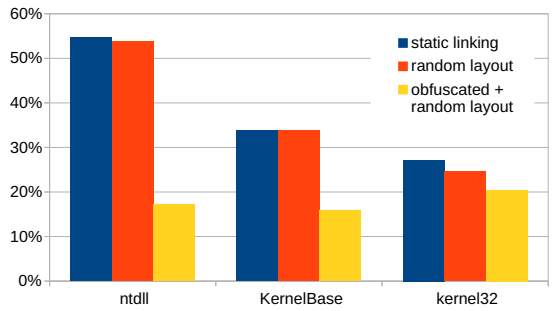


Fig. 4: Percentage of instructions from the different DLLs that BinDiff matches in different statically linked versions of the binary.

extract the necessary information from the original library, where it is present in the ENT.

Figure 4 shows the results of matching each of three DLLs with different statically linked binaries with BinDiff [11]. BinDiff is able to match a significant fraction of the code of ntdll when it has only been statically linked into the binary. Applying layout randomization has little effect on the robustness of the matching. The same holds for KernelBase, but not for kernel32. The large decrease in matching when applying layout randomization is most likely due to IDA Pro incorrectly disassembling some code regions in the binary, which then percolates through BinDiff's iterative matching strategies applied on the constructed control flow graphs. Furthermore, we observe that in all cases applying even a single type of obfuscation already significantly decreases the amount of matched code. In line with our previous research results in the domain of diversification, we expect that combining multiple forms of obfuscations will render diffing tools almost completely useless for attackers [15].

IX. Conclusions

In this paper we presented a proof-of-concept implementation of two techniques to automatically obfuscate the interfaces between Windows applications and the DLLs they link against. The first technique removes the import tables from the binary and has the application itself resolve its required symbols at run time. In the second technique, we statically link the DLLs – including Windows system libraries – into the application and obfuscate the resulting code. We can provide backwards compatibility with previous versions of Windows and evaluated that the protection of static linking combined with obfuscation can to a large extent thwart static diffing attacks.

Acknowledgement

Part of this research was funded by the Fund for Scientific Research - Flanders (FWO) under project grant 3G013013.

References

[1] L. Van Put, D. Chanet, B. De Bus, B. De Sutter, and K. De Bosschere, "DIABLO: a reliable, retargetable and extensible link-time rewriting framework," *Proceedings of the Fifth IEEE International Symposium on Signal Processing and Information Technology, 2005.*, 2005.

[2] y0da, "Lordpe," http://www.woodmann.com/collaborative/tools/index.php/LordPE.

[3] G. Hunt and D. Brubacher, "Detours: Binary interception of Win32 functions," in *Third USENIX Windows NT Symposium*. USENIX, July 1999, p. 8.

[4] mcMike, "WinJect," http://www.cheat-project.com/cheats-hacks/1382/WinJect-1.7/.

[5] dark_byte, "Cheat Engine," http://www.cheatengine.org/.

[6] O. Yuschuk, "Ollydbg," http://www.ollydbg.de/.

[7] Hex-Rays, "IDA Pro," https://www.hex-rays.com/products/ida/.

[8] ——, "IDA F.L.I.R.T. Technology: In-Depth," https://hex-rays.com/products/ida/tech/flirt/in_depth.shtml.

[9] E. R. Jacobson, N. Rosenblum, and B. P. Miller, "Labeling library functions in stripped binaries," in *ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2011.

[10] A. Thierry, "Recognition of binary patterns by morphological analysis," in *Recon*, 2012.

[11] "BinDiff," http://www.zynamics.com/bindiff.html, Zynamics, 2012.

[12] T. Dullien and R. Rolles, "Graph-based comparison of executable objects," in *Proceedings of the Symposium sur la Sécurité des Technologies de l'Information et des Communications*, 2005.

[13] H. Flake, "Structural comparison of executable objects." in *Proceedings of the Detection of Intrusions and Malware & Vulnerability Assessment, GI SIG SIDAR Workshop*, 2004, pp. 161–173.

[14] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," The University of Auckland, New Zealand, Tech. Rep. 148, 1997.

[15] B. Coppens, B. De Sutter, and J. Maebe, "Feedback-driven binary code diversification," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, pp. 24:1–24:26, 2013.

[16] B. Coppens, B. De Sutter, and K. De Bosschere, "Protecting your software releases," *IEEE Security & Privacy*, vol. 11, no. 2, pp. 47–54, 2013.

[17] C. Collberg, "The Tigress Diversifying C Virtualizer," http://tigress.cs.arizona.edu.

[18] C. Liem, Y. X. Gu, and H. Johnson, "A compiler-based infrastructure for software-protection," in *Proceedings of the Third ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, ser. PLAS '08. New York, NY, USA: ACM, 2008, pp. 33–44.

[19] University of Applied Sciences and Arts Western Switzerland of Yverdon-les-Bains, "obfuscator-llvm," https://github.com/obfuscator-llvm/obfuscator/wiki.

[20] M. Madou, "Application security through program obfuscation," Ph.D. dissertation, Ghent University, 2007.

[21] A. Dinaburg and A. Ruef, "Mc-semantics," https://github.com/trailofbits/mcsema.git.

[22] K. Anand, M. Smithson, A. Kotha, K. Elwazeer, and R. Barua, "Decompilation to compiler high IR in a binary rewriter," University of Maryland, Tech. Rep., 2010.

[23] A. Bougacha, G. Aubey, P. Collet, T. Coudray, and A. de la Vieuville, "Dagger," http://dagger.repzret.org/.

[24] Charles Stark Draper Laboratory, "Fracture," https://github.com/draperlaboratory/fracture.

[25] M. Smithson, K. ElWazeer, K. Anand, A. Kotha, and R. Barua, "Static binary rewriting without supplemental information," in *WCRE*, 2013.

[26] PEBundle, "PEBundle." [Online]. Available: http://bitsum.com/pebundle.asp

[27] MoleBox, "MoleBox." [Online]. Available: http://www.molebox.com/

[28] ReWolf, "DLLPackager." [Online]. Available: https://code.google.com/p/dllpackager/

[29] skape, "Understanding Windows shellcode," nologin, Tech. Rep., 2003.

[30] A. Albertini, "Corkami MakePE," https://code.google.com/p/corkami/source/browse/trunk/misc/MakePE/examples/imports/imports_checksum.asm?r=179&spec=svn332.

[31] M. Jurczyk, "Windows X86 System Call Table (NT/2000/XP/2003/Vista/2008/7/8)." [Online]. Available: http://j00ru.vexillium.org/ntapi/

[32] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, , and G. Hunt, "Rethinking the library os from the top down," in *ASPLOS*, 2011.

[33] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the 10th ACM conference on Computer and Communications Security*, 2003, pp. 290–299.