Experiences in Malware Binary Deobfuscation

Hassen Saïdi Phillip Porras Vinod Yegneswaran
Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025, USA
{saidi,porras,vinod}@csl.sri.com
Tel: 1.(650).859.3810

Abstract

Malware authors employ a myriad of evasion techniques to impede automated reverse engineering and static analysis efforts. The most popular technologies include 'code obfuscators' that serve to rewrite the original binary code to an equivalent form that provides identical functionality while defeating signature-based detection systems. These systems significantly complicate static analysis, making it challenging to uncover the malware intent and the full spectrum of embedded capabilities. While code obfuscation techniques are commonly integrated into contemporary commodity packers, from the perspective of a reverse engineer, deobfuscation is often a necessary step that must be conducted independently after unpacking the malware binary. In this paper, we describe a set of techniques for automatically unrolling the impact of code obfuscators with the objective of completely recovering the original malware logic. We have implemented a set of generic debofuscation rules as a plug-in for the popular IDA Pro disassembler. We use sophisticated obfuscation strategies employed by two infamous malware instances from 2009, Conficker C and Hydraq (the binary associated with the Aurora attack) as case studies. In both instances our deobfuscator enabled a complete decompilation of the underlying code logic. This work was instrumental in the comprehensive reverse engineering of the heavily obfuscated P2P protocol embedded in the Conficker worm. The plug-in is integrated with the Hex-Rays decompiler to provide a complete reverse engineering of malware binaries from binary form to C code and is available for free download on the SRI malware threat center website: `http://www.mtc.sri.com/deobfuscation/`.

1 Introduction

There have been substantial efforts in recent years to develop automated tools and services that can reverse malicious binary program logic and profile the behavioral and forensic impacts of malware. In general, the objective of the malware analyst is to extract an understanding of how the malware works (its program logic, control flow, triggering events) and how its presence may be detected or prevented on hosts. Unfortunately, malware binary reverse engineering is a highly adversarial activity. Malware developers share and incorporate a myriad of antianalysis mechanisms to prolong the life of their applications and reduce their detectability. These mechanisms include such protections as code and data segment encryption, binary polymorphism and metamorphism, code restructuring, API call hiding, and antitracing logic. Malware analysis strategies largely explore malware from either of two perspectives: 1) a static review of the application, or 2) an in-depth

monitoring of the application's dynamic behavior. Static program analysis has the advantage of analyzing the entirety of code segments encapsulated in the application, and thus has the potential to capture its complete logical flow. Static analysis can bring to bear in-depth program analysis techniques, such as computing the full control flow graph (CFG) of all code segments, including those segments that may or may not appear reachable. Dynamic analysis involves executing a malware sample in a carefully monitored environment to build a profile of the application's runtime behavior. Tools like CWSandbox [14], TTAnalyze [13], and BitBlaze [6] are examples of online services that perform dynamic analysis. The power of the dynamic analysis approach is that one can rapidly gain a high degree of insight into the runtime operation of a malware sample, including its execution sequence and forensic impact, regardless of what structural transformations have been made to hinder static code inspection. To achieve greater completeness in the runtime exploration of a malware application, dynamic analyses have been applied in an iterative execution form to explore all reachable program statements within the code. Indeed, full path exploration [9] can be used to exhibit all the behaviors while bypassing suicide logic, antitracing, and antidebugging logic embedded in the malware binary. Here, we subscribe to the view that while dynamic analysis has the advantage of bypassing code obfuscation techniques, it is more desirable to examine statically the entire code logic as it is the most reliable way to explore all possible behaviors of the malware. While static analysis is often impeded by the heavy use of various obfuscation techniques, we undertake the task of systematically identifying the various obfuscation techniques applied to a given binary and systematically undoing them to recover the original binary code from which more advanced techniques such as complete code decompilation and code analysis can be applied. A perfect illustration of the power of our approach is the complete analysis [10] of the most obfuscated logic of the Conficker worm [1] that still plagues the world nearly eighteen months after it was first released. In its latest version called Conficker C [2], a heavily obfuscated custom peer-to-peer (P2P) protocol was included in the malware's logic. Dynamic analysis could reveal some features of the protocol such as a partial identification of the message format exchanged between Conficker-infected peers, as well as the number of threads started to handle the P2P traffic. Dynamic analysis, however, did not allow a thorough and precise description of the protocol. This was done only through static analysis where not only a complete description of the protocol including the various message formats, threads and code has been described, but also previously unknown features and functionality were uncovered. For instance, static analysis identified the mapping between the IP address of the infected host and the ports used to channel the TCP and UDP P2P traffic. We were able to extract and deobfuscate the port generation algorithm code [3] from the binary and generate a corresponding C code that we provided to the security community to scan the Internet for Conficker-infected hosts. Most noticeable was the discovery that the P2P protocol not only was used to distribute additional binaries but also was a way to send x86 instructions to be executed as a thread inside the running Conficker process and therefore could serve as a hot patching mechanism that can apply arbitrary modifications to the running process, thus potentially bypassing any monitoring mechanism. Also noticeable was an algorithm that randomly chooses a peer to communicate with among a list of peers composed of randomly generated IP addresses and a list of peers received from a peer-infected host. The algorithm favored heavily the randomly generated list of peers, making efforts of poisoning the list of peers obsolete. This strategy was contemplated by the security community as a way to reduce the size of the Conficker C peer network and therefore contain the infection.

The contribution of our work can be summarized as a set of transformations that

undo systematically the obfuscation steps applied to the original binary. Furthermore, we developed a systematic approach to evaluating any deobfuscation strategy by using decompilation as the evaluation criterion of how well a deobfuscation technique performs. If the resulting code can be translated into a higher-level C-like description, it indicates that the employed deobfuscation technique has restored the binary into a form that a decompiler can make sense of. We take our deobfuscation effort one step further and we provide an automated way of rebuilding the malware binary by identifying the origin entry point (OEP) and the often obfuscated library calls to the Windows application programming interface (API). We have implemented a deobfuscation library as a plug-in for the popular IDA Pro [5] disassembler. We use the Hex-Rays decompiler [8] in our analysis.

The paper is organized as follows. In Section 2, we describe an overview of the obfuscation techniques used by malware and focus on those tackled by our approach. In Section 3, we describe the systematic deobfuscation techniques that we propose. In Section 4, we give an overview of our unpacking strategy. In Section 5, we describe our automated binary rewrite methodology to undo malware obfuscation steps, and in Section 7, we provide a way to evaluate it. In Section 6, we describe our binary rewrite technique, and in Section 9, we describe implementation. Finally, in Section 8 we describe how our techniques are used on Conficker and Hydraq. In Section 9, we describe our implementation. We then conclude with some future research plans.

2 Malware Obfuscation

One major obstacle when it comes to analyzing malware samples collected from the Internet is that the majority of them are packed. Packing is a method commonly used by malware authors to evade signature-based antimalware software and hamper static analysis. It involves compressing (or encrypting) most or all of the malicious code and then wrapping the result in an often short set of instructions that implements an unpacking routine that decompresses (or decrypts) the packed code at runtime and then transfers control to the original unpacked code. Malware obfuscation is often employed as an important step in the packing process. There are mainly two types of packing technologies. Both technologies use some form of encryption and code compression to reduce the footprint of the malware. They are, however, distinguished by the transformations operated on the original binary code. The first type of packing technology transforms the x86 code into an equivalent form where a myriad of transformations has been employed to obfuscate features of the code. These features include the control flow graph, the call graph, the import table, and the flow of data. This produces a functionally equivalent code that is much harder to analyze manually and represents a challenge to reverse engineers. The second type of packing technology consists of hiding the behavior of the malware by emulating the code. Portions of the malware code are moved to a data portion of the binary and are emulated within an embedded virtual machine that executes instructions hidden in the data portion and dispatches the control flow appropriately to mimic the original code functionality. In this type of technology, the resulting code is a mix of the original binary code and an emulation code that fetches the obfuscated instructions from the data portion of the binary and emulates them along with a dispatching routine that emulates the control flow graph of the original binary. A full taxonomy of the various obfuscation techniques can be found in [7]. In this work we are particularly interested in transformations that produce a semantically equivalent program that has a different structure than the original program. The following describes the categories of obfuscations that we

tackle:

Packing: The malware's code is compressed or encrypted in order to minimize the malware's footprint and to also generate multiple copies of the same malware.

Antianalysis techniques: Various antidebugging and virtual machine detection instructions are inserted into the original binary to trigger suicide logic and prevent the dynamic tracing of the malware.

Binary rewriting: The malware's instructions are rewritten to semantically equivalent instructions that cannot be attributed to a particular compiler and do not correspond to the product of a compilation process. These rewriting steps can target the control and data flow, the functions epilogues and prologues, stack manipulation, and calling conventions.

API obfuscation: the obfuscation of the library or Windows API calls made by the malware is achieved by destroying the original import table and rebuilding it on the fly as part of the unpacking routine.

3 Deobfuscation

We systematically address the above-mentioned obfuscation categories, but while many approaches have been proposed in the literature to address particular techniques, we focus on the metrics to evaluate the quality of deobfuscation to ensure that we have successfully recovered the original code. Malware binaries are often the product of a compilation process where symbol tables and debugging symbols are omitted. Further packing steps compress the code and destroy the import table, which is later restored dynamically. Obfuscation is employed either as a step prior to packing where the binary code is transformed into a semantically equivalent binary with a different structure, or employed in tandem with the packing where not only the binary is transformed into an equivalent program but where checks are inserted to make sure that the program is not monitored or being debugged, which effectively challenges both dynamic and static malware analysis techniques. Knowing that the original malware binary prior to obfuscation is the product of a compiler and therefore adheres to particular structural conventions such as function epilogues and prologues and calling conventions, we aim at undoing the obfuscation steps employed by the malware authors by rewriting the code into a form that does correspond to what a typical compiler would produce. To ensure that our deobfuscation is successful, we first need to recover the binary code through an unpacking process that defeats the antitracing and debugging techniques embedded in the packed binary. We also need to recover the OEP and rebuild the import table before rebuilding the executable binary. We undertake the following main deobfuscation steps we :

Unpacking: Unpacking malware amounts to running the malware and capturing its process image and then writing it to an executable file format that can be subjected to further analysis such as disassembly and code analysis. Unpackers are usually one of two kinds: generic unpackers that focus on process memory image dumping without addressing the problem of import table recovery, and those that are dedicated to a particular packer and thus are able in some cases to automatically rebuild import tables and discover the OEP. We devise a multistrategy for dumping a process image of the malware while bypassing the antitracing and antidebugging techniques embedded in the malware unpacking routine.

Binary rewriting and editing: We undo any rewriting step that has been applied to the original binary code after the compilation process of the original malware source code. We validate the result through decompilation to make sure that the result corresponds to a well-formed C-like code.

Malware binary reconstruction: We attempt to reconstruct the original unpacked malware executable by setting the OEP and rebuilding the import table. We use a semantic and a structural approach to analyzing the dumped process image of the malware to determine the origin entry point. We employ a set of heuristics to identify API calls and rebuild the import table accordingly.

4 Malware Unpacking

Malware unpacking refers to the process of recovering the original code from a packed malware binary. This process involves running the malware binary and capturing its process image. The image process contains both the unpacking routine and the original code revealed at runtime. The original code can be either the product of a compiler or a rewrite of the product of a compiler in order to obfuscate the malware logic. Analyzing the image process, or what we refer to as the *raw dumped binary*, is the objective of any reverse engineering effort - that is, capturing the original binary code and understanding its logic and purpose. While this is possible in many instances, there are cases where the malware has been packed in such a way that the original code cannot be fully recovered. This is the case where the original code is weaved with additional code inserted by the packer to check for a debugger or other tracing methods, or when the code is emulated by embedding an emulator in the malware binary.

Our first step is to dump the process image of a running malware binary while bypassing all antitracing and antidebugging logic, otherwise called *antidumping logic*, embedded in the binary. To achieve this, we employ a multistrategy that allows us to try several heuristics until all antidumping techniques are circumvented. The following describes the different strategies we use:

- Nonintrusive monitoring techniques allow a coarse-level tracing of native kernel-level API or system calls, and then dumping of the malware's process image. We use our unpacker Eureka [12] for this purpose. Eureka dumps the memory image when the `NtExistProcess` system call is executed. It can dump the memory image when a certain number of occurrences of binary n-grams is reached. It can also dump the memory image after a user-defined timer expires.

- If system call tracing is detected, we proceed by running the malware and then suspend the execution by attaching a debugger to the running process before dumping it. This allows suspension of the process without actively tracing it with a debugger. Process suspension allows us to examine the memory and dump the appropriate memory segments of the running process.

- If the OEP is known, we turn on system call monitoring only when the OEP is reached to monitor the execution of the original code. This ensures that the antitracing and antidebugging techniques generally employed before transferring the control to the OEP will never be triggered.

Once an image of the running process is captured, it is written to a file that is a Portable Executable (PE) format file or `.exe`. The file is then subjected to further

analysis that consists of finding the OEP, identifying and reconstructing the import table, and undoing any other obfuscation step that resulted in rewriting the original set of instructions.

5 Malware Binary Deobfuscation through Code Transformation

Like any other binary, a malware binary is the product of compiling the source code into machine instructions for the given target platform. Analysis tools such as disassemblers can then recover the assembly code by reinterpreting the binary code into a higher-level language. Decompilers take this process one step further and translate the assembly code to source code. Compilers often produce binary code in a form from which it is possible to guess the type of compiler used and therefore can often determine the programming language and the programming platform used to compile the original malware code. Obfuscated code often does not exhibit these clues since it is the product of a systematic rewrite of the original binary code into a semantically equivalent form but whose structure has been altered. Among the most used obfuscation techniques through systematic rewrites are

1. Rewriting stack manipulation instructions such as push and pop into `mov` and `add` and `sub` instructions where only the ESP pointer is increased or decreased. This prevents the disassembler from recognizing access to the stack frame and recognizing read or write operations of the local variables and access to function arguments.

2. Code dechunking where a single subroutine composed of multiple contiguous blocks is scattered throughout the binary image and where the control flow between contiguous blocks is ensured through an unconditional jump instruction. This effectively dechunks a function into multiple chunks where chunks present in multiple subroutines are represented by a single copy referenced by multiple subroutines. This prevents the disassembler from recognizing function boundaries.

3. Calling convention obfuscation where the standard calling conventions are replaced by user-level conventions. Calling conventions describe the interface of called code - that is, the order in which parameters are allocated, where parameters are placed (pushed on the stack or placed in registers), which registers may be used by the function, and whether the caller or the callee is responsible for unwinding the stack on return. By pushing some of the arguments on the stack and passing the remaining one through an arbitrary set of registers, it is possible to define arbitrary user-level calling conventions that do not correspond to the output of any standard compiler. This prevents the disassembler from guessing the number of arguments for a given function and recognizing the calling convention which might indicate which compiler was used.

We proceed in a systematic way to undo the above-mentioned obfuscating techniques by applying a set of transformations. Our transformations aim at rewriting the binary in a form that corresponds to the output of a standard compiler. To evaluate how well our transformations work, we apply systematically a decompiler to the resulting binary to ensure that the decompiler can recognize the output of a compiler and therefore is able to translate the assembly code into a higher level of abstraction in the form of a C-like program. The set of transformations we apply is

1. Normalize the stack pointer manipulation through the use of push and pop.

2. Identify all chunks that belong to a function and then apply decompilation to recover the C-like description of each function in the unpacked malware binary.

3. Identify nonstandard calling conventions and systematically translating them to the `cdecl` standard calling convention.

Given that our transformations produce an assembly code that can be successfully translated into a C-like set of functions, we proceed to identify the API calls and the OEP to complete our binary deobfuscation and reconstruction process.

6 Malware Binary Reconstruction

In malware reconstruction, a raw dumped image process is rebuilt by discovering and statically rebuilding the import table as well as the OEP. Once this has been achieved, it is possible to redisassemble the dumped executable with the help of the additional recovered information, which allows the disassembler IDA [5] to produce a better disassembly. Figure 1 shows the disassembly of a function in a damaged binary prior to OEP and API identification. Figure 2 shows the same function after the OEP and all API calls were identified. Providing this information to IDA allows the disassembler to recognize that the function is `WinMain` and to recognize all of its arguments and local variables accurately. Finding the OEP and achieving a 100% API resolution rate are keys to the overall success of the malware binary reconstruction process in particular and to the overall goal of reverse engineering and analysis. Finding the OEP and the target of every call in the code allows us to generate the complete call graph and the reconstruction of the import table, which then allows us in the best-case scenario to produce an unpacked code that can actually be executed without the obfuscation introduced by the packer. In the worst case, we can still build a complete call graph and control flow graph but we may not be able to execute the original code.

6.1 Finding the OEP

Given the disassembly of a dumped image of a running malware binary, we employ two distinct and complementary strategies to identify the OEP.

6.1.1 Structural Strategy for OEP Discovery

We build a call graph using the disassembly of the damaged binary by extracting from each subroutine the set of subroutines that are referenced at any address in the subroutine address space. The graph might have several root nodes with a different number of callee subroutines. We rank the root nodes by ascending number of called subroutines, and we determine that the OEP is one of those root node functions. The following example illustrates how the OEP candidate is determined for a version of the Storm worm [11]:

```
checking for OEP candidates using graph connectivity
Found the following OEP candidates with 435 successors ...sub_403318
Found the following OEP candidates with 97 successors ...sub_40BEC2
Found the following OEP candidates with 83 successors ...sub_403C39
```

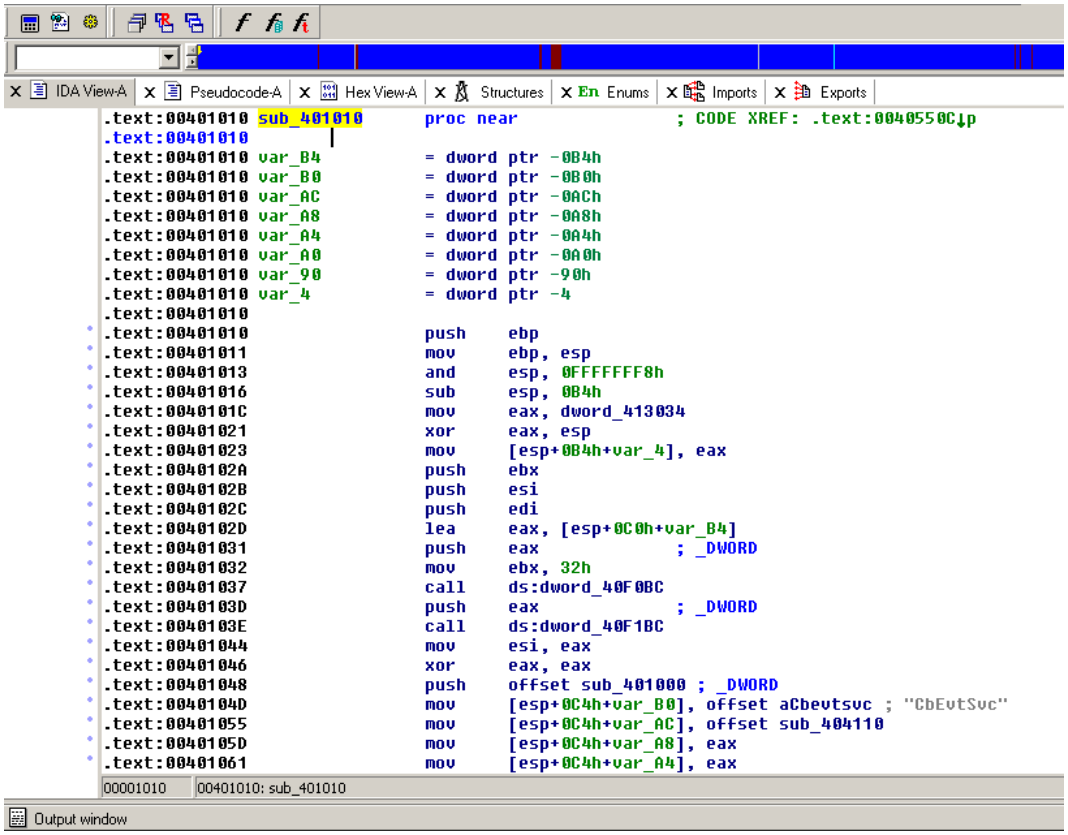

Figure 1: A Function before OEP and API identification

The subroutine at address 0x403318 is the subroutine that is the most likely candidate for OEP. The subroutines `sub_40BEC2` and `sub_403C39`, while not referenced directly from subroutine `sub_403318`, are determined later to be the start addresses of two threads started from the main thread of the executable starting at address 0x403318.

6.1.2 Semantics-based Strategy for OEP Discovery

Since we are dealing with Windows executables, we exploit the fact that the malware executables have been compiled as Windows applications having specific features. It is often the case that at the beginning of the execution of a Windows application, a certain number of known Windows APIs are invoked. For instance:

- `GetCommandline` is often called at the beginning of a Windows executable to retrieve command line arguments.
- `GetModuleHandle` is invoked to retrieve a module handle for a specified module loaded by a calling process or the handle to the file used to create the current process.
- `GetVersion` retrieves the version number of the current operating system.
- `CreateMutex` is often invoked in the beginning of the execution to check whether a version of the malware is already running on the host.

- `ExitProcess` is often called in the start subroutine where the malware executes its logic and exits or checks for specific resources on the local host and then exits prematurely when those resources are not present, or checks for antitracing and debugging mechanisms and then exits prematurely (suicide logic).

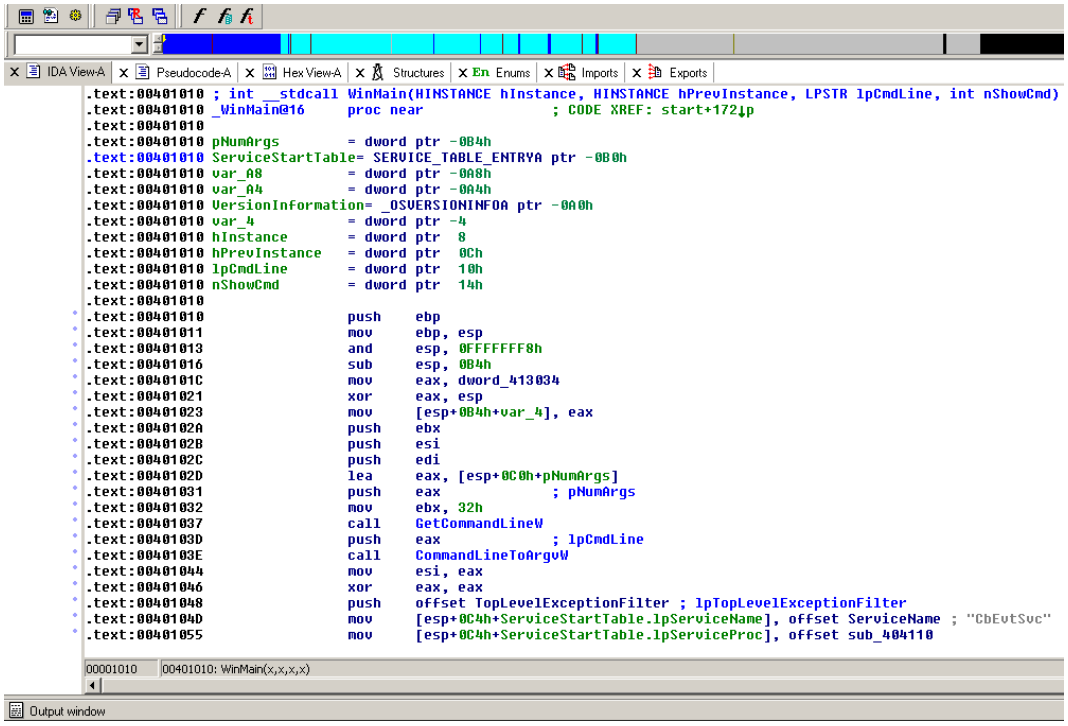

Figure 2: A Function after OEP and API identification

We have determined that there are more than twenty APIs or a combination of those that are a very good indication of the start of a Windows executable. Combined with the call graph analysis, we achieve a 100% rate of OEP discovery by simply analyzing the disassembly of raw dumped processes. Using the right OEP, IDA can improve its disassembly process by better determining the boundaries of functions, better tracing references to functions resulting in better call graphs, and better identification of subroutines that correspond to the C runtime library. This yields a disassembly where on average 75% of the code is identified as known runtime libraries that have precise semantics and do not need to be included in the summary of the malware's behavior. This means that we can focus on average on only 25% of the original malware's code, which corresponds to malware-specific behavior such as changes to the registry, changes to the file system, launching and terminating processes, and remote file download and network connections.

6.2 API Resolution

In much of the malware unpacking literature, the recovery of import tables is not addressed. The Eureka framework [12] was the first to address such concern by trying to identify Windows APIs when the jump target is given as an absolute address - that is, an absolute address that is known to be the standard address where a given Windows API is always loaded. The typical Windows XP installation with a service pack includes about 1283 DLL files in the system32 directory. Adding to those standard libraries, Windows

stores in the Winsxs folder multiple versions of DLLs in order to let multiple applications run in Windows without any compatibility problem. In XP, few DLLs have multiple versions, leading to a total number of DLLs in the system with their multiple versions to be about 1350. Windows Vista and 7 maintain a much significantly larger number of copies of the standard DLLs. We have built a database of all exported DLL functions for a typical XP installation with SP2 and SP3. All APIs are identified by a name, the number of arguments, the name and type of each argument when known, the type of the return value, and the address where the API is typically loaded - that is, the offset of the API location with respect the DLL file. We extended extend the capabilities of Eureka based on the documented APIs. When determining that a call instruction is to an address that corresponds to the standard location of an API, we match the call to the identified API. When an absolute address is not give and it is not possible to map a call to a particular API, we resort to a novel API resolution technique based on type analysis. We first analyze the number of elements pushed onto the stack prior to reaching the call site. This allows us to determine the number of arguments for the target function. We then apply data flow analysis to trace how the return values of the unidentified target functions are related to arguments of other unidentified functions. The union of these constraints is matched against our derived type signatures of all candidate Windows APIs that we documented in our database. Malware can load dynamic-linked libraries (DLLs) in nonstandard locations. The packer MEW, for instance, replaces all import entries with addresses of user-level functions that happen to have the same signature as the obfuscated APIs. The user-level function address is replaced by the Windows API address only when the API is invoked. In the case of packers like Themida, ASPack and Armadillo for instance, the import table is built in two phases and only some of the APIs are referenced by their absolute address in a static manner. All other references are dynamically computed. Only Type analysis can identify all APIs.

7 Using Decompilation

Decompilation is the reverse operation to that of a compiler. That is, it translates the low-level assembly representation of a binary file into a higher level of abstraction to be humanly readable and that is in some cases very close to the original source code of the program prior to the compilation phase. The term *decompiler* is most commonly applied to a program that translate executables programs (the output from a compiler) into source code in a (relatively) high-level language that, when compiled, will produce an executable whose behavior is the same as the original executable program. The Hex-Rays [8] decompiler for instance, produces for each subroutine in a binary, the corresponding C-like function. The decompiler identifies the argument of the subroutines, its local variables, and its return value. This effectively determines the signature of the function and its calling convention. The decompiler then translates the assembly instructions into C-like assignment expressions where all references of x86 registers used as intermediate storage and computation variables are eliminated. It also translates jump instructions into C control constructs. The success of decompilation depends on the amount of information present in the code being decompiled and the sophistication of the analysis performed on it. Some post-compilation tools produce obfuscated code (that is, they attempt to produce output that is very difficult to decompile). This is done to make it more difficult to reverse engineer the executable.

Figure 3 shows the result of decompiling a subroutine in the Hydraq binary. The decompiler was fooled to believe that the function takes 57 arguments. This is because

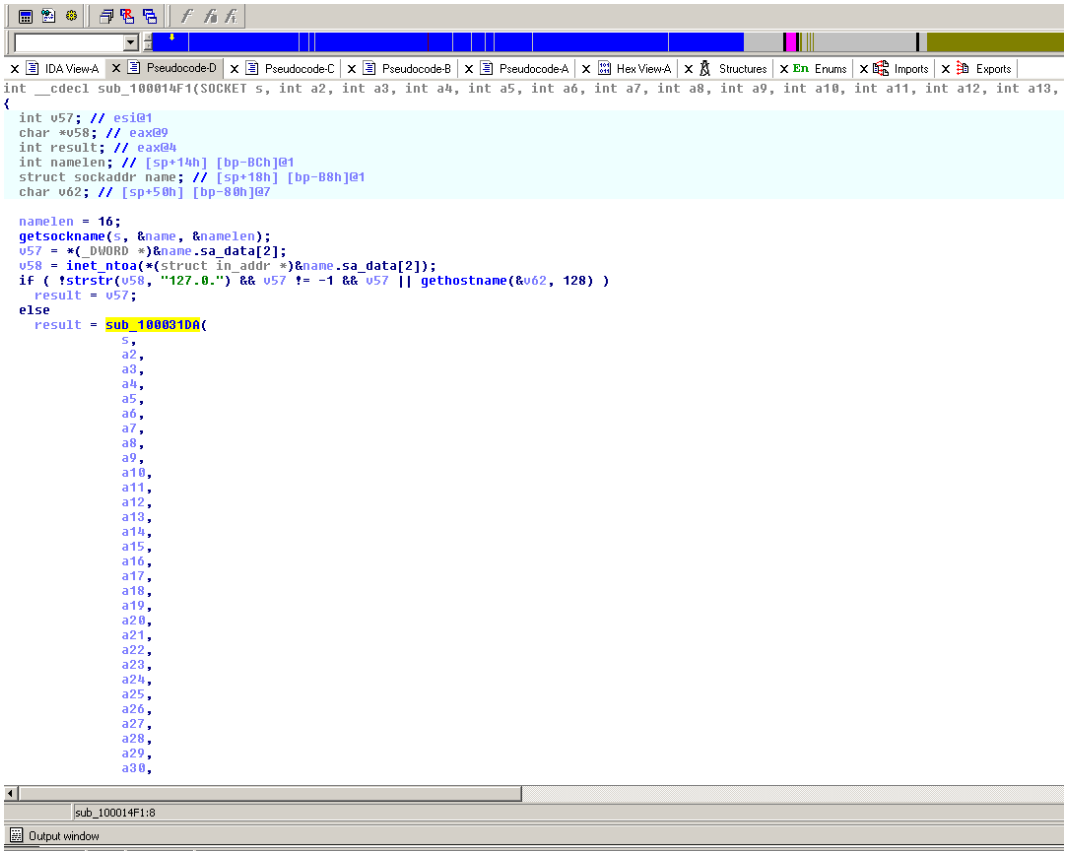

Figure 3: Obfuscated Function at Offset 0x14F1 in Hydraq

the return value of the subroutine is determined by a different subroutine that was not decompilable as a result of dechunking. Figure 4 shows the result of decompilation of the same function after we merge the two functions into a single one as a systematic way of recovering from dechunking. Notice that the decompiler successfully identifies a single arguments and all local variables.

8 Deobfuscating Conficker

Each generation of Conficker [1] has incorporated techniques such as dual-layer packing, encryption, and antidebugging logic to hinder efforts to reverse its internal binary logic. Conficker C [2] further extends these efforts by providing an additional layer of cloaking to its newly introduced P2P module. The binary code segment that embodies the P2P module has undergone multiple layers of restructuring and binary transformations in an effort to substantially hinder its reverse engineering. These techniques have proven highly effective in thwarting the successful use of commonly used dissassemblers, decompilers, and code analysis routines employed by the malware analysis community. In particular, three primary transformations were performed on the P2P module's code segment:

API Call Obfuscation: Conficker employs a common obfuscation technique, in which library references and API calls are not imported and called. Rather, they are often replaced with indirect calls through registers in a manner that hides direct insight into which libraries and APIs are used within a segment. As API and library

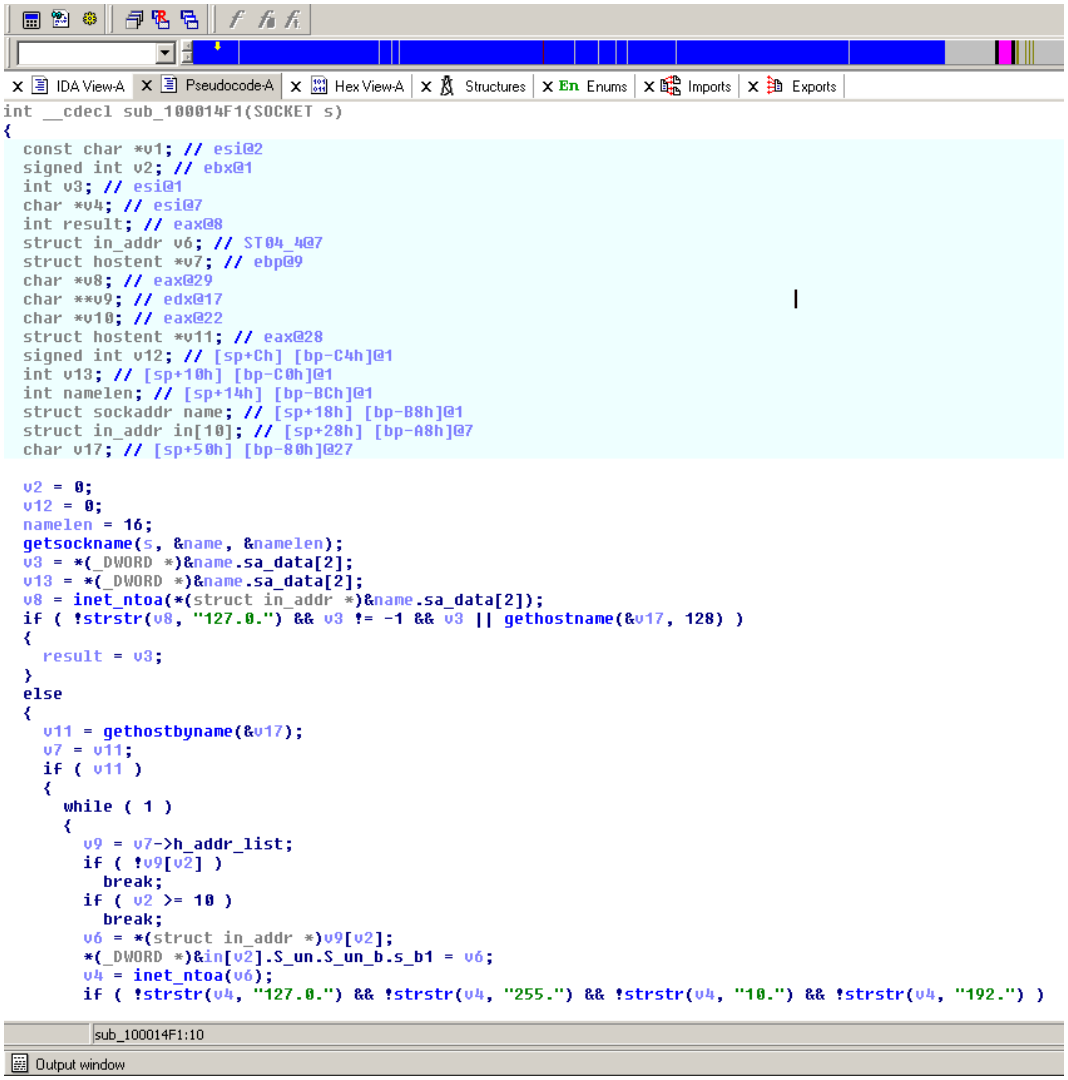

Figure 4: Deobfuscated Function at Offset 0x14F1 in Hydraq

call analyses are critical for understanding the semantics of functions, loss of these references poses a significant problem to code interpretation.

Control Flow Obfuscation: The control flow of Conficker's P2P module has been significantly obfuscated to hinder its disassembly and decompilation. Specifically, the contents of code blocks from each subroutine have been extracted and relocated throughout different portions of the executable. These different blocks (or chunks) are then referenced through unconditional and conditional jump instructions. In effect, the logical control flow of the P2P module has been obscured (spaghettied) to a degree that the module cannot be decompiled into coherent C-like code, which typically drives more in-depth and accurate code interpretation.

Calling Convention Obfuscation: Decompilers depend on their ability to recognize compilation convention such as function epilogues and prologues. Such segments help the decompiler interpret key information, such as calling conventions for each subroutine, which in turn enable the decompiler to interpret the proper number of

function arguments and local variables. Unfortunately, the P2P module has been transformed to disrupt such interpretations. Each subroutine has been translated such that some parameters are passed through the stack using push instructions, while others are passed by registers, and in unpredictable order. In effect, these transforms utterly confuse decompilation attempts, generating inaccurate function argument and local variable lists per subroutine. In the presence of such errors, code interpretation is nearly futile.

Due to these obfuscations, the resulting source code derived from the decompiler incorporates fundamental misinterpretations that hinder semantic analyses. We have been able to systematically undo all the binary transformations in the P2P module, and have produced a full decompilation of this module to a degree that approximates its original implementation. The full code is available in [10]. We identified all 88 obfuscated APIs. We normalized all calling conventions to the `cdecl` calling convention. We then systematically undid the dechunking obfuscation for all functions and managed to decompile properly each of them. Unlike the Conficker P2P logic, Hydraq [4] did not exhibit the same level of obfuscation. It did, however, share some obfuscation features with Conficker. The functions of the Hydraq binary have been subjected to dechunking, which renders decompilation difficult. We applied our transformations to automatically generate the C-like code for each subroutine and build a complete CFG of the binary. The IDA disassembler identified 185 subroutines in the binary prior to our analysis. After running the dechunking transformation, only 158 subroutine remained and were decompiled. Our analysis allowed us in many instances to merge several subroutines into a single one.

9 Implementation

We have implemented our deobfuscation techniques into an IDA plug-in that is available for versions 5.X of the popular disassembler. After using the plug-in, it is possible to invoke the Hex-Rays decompiler [8] to validate the result of deobfuscation.

10 Conclusion and Future Work

We have presented a set of techniques for automatically undoing the work of obfuscators to help with the reverse engineering of malware. We have applied our techniques to two highly publicized examples of malware: Aurora and Conficker. We have shown that our techniques in the case of the heavily obfuscated Conficker P2P protocol can uncover the entire logic of the protocol, therefore enabling reverse engineering of the worm. Our work is readily available as a plug-in for the popular IDA disassembler to assist reverse engineers in their task. We plan on extending our work to handle the increasingly used virtualization and emulation, and on exploring automated ways to detect and undo wider classes of obfuscation techniques.

Acknowledgment

This material is based upon work supported through the U.S. Army Research Office under the Cyber-TA Research Grant No. W911NF-06-1- 0316. The views expressed in this document are those of the authors and do not necessarily represent the official position of the sponsors.

References

[1] http://en.wikipedia.org/wiki/Conficker.

[2] http://mtc.sri.com/Conficker/addendumC/.

[3] http://mtc.sri.com/Conficker/contrib/scanner.html.

[4] http://en.wikipedia.org/wiki/Operation_Aurora.

[5] IDAPro Dissasember . http://www.hex-rays.com/idapro/.

[6] D. Brumley, C. Hartwig, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, D. Song, and H. Yin. Bitscope: Automatically dissecting malicious binaries. CMU Technical Report, 2007.

[7] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, July 1997. http://www.cs.auckland.ac.nz/~collberg/Research/Publications/.

[8] Hex-Rays. The Hex-Rays Decompiler. http://www.hex-rays.com/decompiler.shtml, 2009.

[9] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 231–245, Washington, DC, USA, 2007. IEEE Computer Society.

[10] P. Porras, H. Saidi, and V. Yegneswaran. Conficker c p2p protocol and implementation. Technical report, SRI International, 2009. http://mtc.sri.com/Conficker/P2P/index.html.

[11] P. Porras, H. Sadi, and V. Yegneswaran. A multi-perspective analysis of the storm (peacomm) worm. Technical report, Computer Science Laboratory, SRI International, October 2007.

[12] M. Sharif, V. Yegneswaran, H. Saidi, P. Porras, and W. Lee. Eureka: A framework for enabling static malware analysis. In *ESORICS '08: Proceedings of the 13th European Symposium on Research in Computer Security*, pages 481–500, Berlin, Heidelberg, 2008. Springer-Verlag.

[13] U.Bayer, C.Kruegel, and E.Kirda. Ttanalyze: A tool for analyzing malware. In *EICAR*, 2006.

[14] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security and Privacy (Vol. 5, No. 2)*, March/April 2007.