Exploiting Code Mobility for Dynamic Binary Obfuscation

Paolo Falcarin[1], Stefano Di Carlo[2], Alessandro[2] Cabutto[2], Nicola Garazzino[2], Davide Barberis[2]
University of East London Computing, United Kingdom[1]
Politecnico di Torino, Torino, Italy[2]
falcarin@uel.ac.uk, stefano.dicarlo@polito.it, cabutto@polito.it, nicola.garazzino@polito.it, davide.barberis@polito.it

Abstract

Software protection aims at protecting the integrity of software applications deployed on un-trusted hosts and being subject to illegal analysis. Within an un-trusted environment a possibly malicious user has complete access to system resources and tools in order to analyze and tamper with the application code. To address this research problem, we propose a novel binary obfuscation approach based on the deployment of an incomplete application whose code arrives from a trusted network entity as a flow of mobile code blocks which are arranged in memory with a different customized memory layout. This paper presents our approach to contrast reverse engineering by defeating static and dynamic analysis, and discusses its effectiveness.

1. Introduction

Software protection aims at protecting the integrity of data and software applications deployed on un-trusted hosts and being subject to illegal analysis. Software Protection is an important requirement for software companies as many of them are increasingly adopting tools with the intention of defending their intellectual property from unauthorized reuse by competitors, i.e. to protect their products against unauthorized reverse engineering, software *cracks* and piracy.

The attacker has no restriction on the tools and techniques to use to reverse-engineer and then to tamper with the application (e.g., super-user privileges are assumed to be available to the attacker). He/she can install any software on the target machine (e.g., debuggers, emulators), to read and write every memory location, processor registers and files. Being in control of the target computer, the attackers can mount *environmental attacks* in which the program will be executed. System libraries and general purpose libraries are controlled by the attackers, along with the operating system. As a consequence, the attackers can use system calls, the input/output subsystem, the network stack, and the memory management subsystem for their purposes.

Attackers possess tools for manipulating binary files at a higher conceptual level than strings of bits, tools that understand file formats and relationships between files. They are assumed to use tools that enable them to transform programs between different formats and between different levels of abstraction (disassemblers, decompilers).

To contrast such attacker goals, it is important to defeat both tools used to perform static analysis (disassemblers, decompilers) and debuggers used for dynamic analysis. To reach this objective, this work exploits the use of code mobility to increase reverse engineering complexity in order to make harder for an attacker understanding the application structure and behaviour. To address this research problem, we propose a novel binary obfuscation approach based on the deployment of a mainly incomplete application whose code arrives from a trusted network entity as a flow of mobile code blocks which are disposed at run-time in memory with a different customized memory layout. The trusted server placed over the network is responsible of deciding the customized memory layout of the binary code that will be sent, block after block, to the application. This paper presents our approach to contrast reverse engineering by defeating static and dynamic analysis, and discusses its effectiveness. The paper is organized as follows: first we describe the problem and state of the art solutions are introduced; then we describe how to enforce software protection with mobile code; finally the effectiveness of our approach is discussed and compared with related works before we draw our conclusions.

2. State-of-the-art

The problem of executing software in an untrusted computing environment has recently gained considerable attention. The literature can be divided into either hardware-based or software-based solutions. The Trusted Computing Group is defining a set of standards to address the problem of executing software in a trustworthy computing environment from a hardware perspective. Sailer et al. [15] build software protections on top of a tamper-proof hardware component, e.g., the Trusted Platform Module [4], which is situated locally on the motherboard. The problem with hardware components is that they cannot be replaced in case of design errors and they may require an expensive . The trustworthiness they provide, covers the machine as a whole (including BIOS and OS) and cannot be granted at a fine-grained level, e.g., for selected applications. Moreover, the integrity verification method is performed off-line and it reacts after the fact.

Many software-based protection techniques have been proposed in latest years both to prevent reverse engineering and code analysis (like obfuscation), or to detect at run-time if the program integrity has been violated by means of additional code bundled in the application. These techniques aim at producing *tamper-resistant* applications.

Obfuscation aims at increasing the attack complexity by making it hard for the attacker to comprehend the behavior of a decompiled program [9].

Obfuscation techniques are based on the addition of complexity to the source code structure (without changing its behavior) through different kinds of code transformations both regarding program's control flow and/or data structures [9]. However, Barak et al. [6] showed that some functions cannot be obfuscated, and other papers claim that perfect obfuscation is impossible. In most cases, breaking obfuscation is just a matter of time and attacker's skills.

Binary obfuscation techniques have been recently proposed to increase reverse engineering complexity: Linn et al. [14] proposed a tool for inflating binary code with redundant and/or *garbage* instructions to defeat disassemblers or to produce a very complex assembly code: they evaluate obfuscation strength with their *confusion factor*, as the percentage of instructions not correctly disassembled because of binary obfuscation. Kanzaki et al. [13] used self-modifying binary code to defeat static analysis and disassembling, while Birrer et al. [7] provide metamorphic binary code by means of program fragmentation.

Code obfuscation transformations are also employed to hide other kind of protections embedded in the software (like tamper-resistant code) so that it cannot be easily detected and removed. Tamper-resistant code aims at identifying attacks like unexpected binary modifications and typically react by stopping the application. Some of these protections rely on an external source of trust, like a locally bundled secure hardware or a trusted network server.

Protection schemes going beyond obfuscation have been proposed but no one so far provides absolute protection. It is therefore highly recommended to complement each protection technique with obfuscation, to increase the expected expiration time of a protected version of a program.

The pioneering work of Aucsmith [5] was proposed to resist to code observation: his technique to break a binary program into individually encrypted segments, so that the hash value of a block is the secret key for decrypting the next block; if the program was altered the hash value is changed and then the next block cannot be decrypted properly and the program cannot continue to run. In this case finding the first key allows to recover the full chain of keys.

Other techniques that can be strengthen by obfuscation include: integrity self-checking, customization, self-modifying code and mobile code.

Customization creates many different copies from an initial version of a program. Each copy of the protected program is different in its binary shape, but is functionally equivalent to other copies [5]. Thus, attacks designed to work with one version might not work with other customized versions. This kind of protection discourages diffusion of *cracks* but it does not aim at detecting and reacting to tampering. More recent research works use self-modifying code, or mobile code to thwart static analysis. *Self-modifying code* [5, 11], at binary level, defeats static analysis and increases the difficulty of dynamic analysis.

Mobile code approaches are only applicable on client-server applications where parts of the binary code (containing both application logic and protection code) are downloaded at run-time from a trusted server: some works provide remote integrity attestation using mobile code on modified JVMs with dynamic AOP [10] or by natively extending JVM 5 through its JVM Tool Interface [16].

3. Software Protection by Mobile Code

To counter reverse engineering, current protections often rely on obfuscation and/or on software-based tamper-resistance techniques relying on code checkers whose position is hidden in the application. However, we observe that any technique that allows the attacker using static analysis is not robust enough. Indeed, code-checkers can be eventually identified and inhibited by an attacker with enough knowledge, time, and reverse engineering tools. Even in presence of binary obfuscation some tools [2] can transform and clean the binary code to remove protections in few days, as shown by the T2 challenge proposed yearly to the reverse engineering community [3]. To overcome the drawbacks of local protection techniques, network-based techniques can be applied. In this scenario a trusted entity placed on the network, and out of the control of the attacker, is in charge of monitoring the execution of the application to protect, and together with dynamic code replacement, reverse engineering attempts can be made more complex by forcing the attacker to continuously face different versions of the program.

The main idea, highlighted in this paper, is to use code mobility to make it more difficult for an attacker to tamper with the code. In particular, code mobility is exploited to create different customized versions of a given program. These versions can be different in space for their different binary structure and in time since during
the execution, in a particular point in time, only a subset of the binary code is actually stored in the client host's memory. Mobility can be therefore used to reduce the visibility on the whole binary code thus limiting the attacker's knowledge and contrasting static analysis.
Code mobility shows many features which are helpful to improve tamper resistance:
• Protection of code against static and dynamic analysis, as the whole code is not completely available when running on the hostile host;
• Application structure behavior is not-predictable as it is decided by the trusted server and customized for every execution;
• Single instance dependency: it is unfeasible to create a custom crack for each different installed copy;
• Easy possibility of extending the architecture with new protection techniques.

The Figure 1 depicts a possible application of code mobility to implement a tamper-proofing architecture.

An application P is deployed to the final user as an almost empty box, containing an empty code section where to place blocks of code and a Binder able to receive these blocks and to map them into the code section thus managing the overall program execution. The trusted entity is a complete secure machine or device placed somewhere on the network. With completely secure we intend that an attacker has no way to tamper with this machine, and moreover it does not know anything about the services running in it.

The network communication between the trusted server and the program to be protected (running on the remote host), created through a network socket, includes two log¬ical channels: a bidirectional control channel used to ex¬change control information, and a unidirectional channel

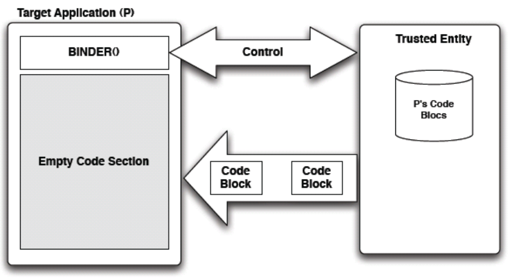
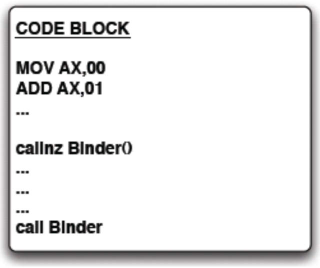


Figure 1. Tamper-resistant architecture with Mobile Code and Replacement

used to send blocks of code to the program. We reuse and adapt the ISO Symmetric Key Three-Pass Mutual Authentication [12] protocol to guarantee mutual authentication between the trusted server and the un-trusted client during start-up phase. When the connection is established with this protocol, the trusted and the un-trusted node can start to communicate. In order to encrypt the communication between the two nodes, and therefore to prevent man¬in-the-middle attacks, the channel have been secured using the AES encryption algorithm. The message is finally signed through the MD5 algorithm for better performance. With the signature of the message, the receiver (the trusted server) can be sure that the message was sent by the correct node. If the signature is incorrect, then the message has been hacked and the flow of code blocks should be interrupted.

The clean program (on the trusted node) will be split in code blocks. The Figure 2 shows an example of code block. Any time the application needs to jump outside the block, either because the execution reaches the end of the block or, control flow instructions need to modify the sequential execution flow, a call to the binder is inserted.



```
CODE BLOCK

MOV AX,00
ADD AX,01
...

callnz Binder()
...
...
...
call Binder
```

Figure 2. Program block example

In order to continue the execution, each time it is called the binder has to:

• Retrieve the position inside the block where the call was issued (this is always possible looking at the application stack);

• Send this information to the trusted entity that will in turn calculate the next block, and the position in the next block where the execution should restart;

• Wait for the transmission of the target block if not already present. Every time a block is sent to the binder, its target location in the code section is decided by the trusted host (e.g., randomly) and sent to the program through the control channel. This step is crucial to make sure that, for every execution of the application, and for multiple executions of the same block during a single execution of the application, the memory layout will be continuously different thus reducing the effectiveness of both static and dynamic analysis. This translates into the fact that the binder does not contain any fixed information about the structure of the program that can be statically analyzed by an attacker. All information are dynamically generate by the server at run-time.

Bogus blocks can be periodically sent to the program in order to continuously confuse the attacker, and to overwrite portions of the empty code section thus reducing the time the attacker has to understand a given portion of code. Bogus block may include, unelectable blocks that generate errors when executed (e.g., they contain illegal microprocessor instructions), and no-effect blocks containing code performing computations that do not produce any useful result for the program.

3.1. Binary Code Instrumentation

In order to implement the proposed program execution schema it is mandatory to be able to split a program binary into a set of different blocks, and to instrument each block in order to insert calls to the binder. The Figure 3 shows the automated instrumentation flow able to start with a standalone application and to automatically generate the related pool of code blocks. The generation is tuned by a set of parameters aiming at defining the optimal length of the blocks to avoid the generation of blocks that are too small or too big.

To instrument a program we first need to interpret its assembly code. This is particularly important to identify control flow instructions that need to be properly managed. This first step can be efficiently performed using a disassembler tool. Disassemblers provide a text description of a binary application easier to be processed. In this work we considered the Intel instruction set architecture.

Given the disassembled code of the program we need to split it into Code Blocks (CBs). A code block is a sequence of contiguous assembly instructions. We can define different approaches to split the code segment into CBs. Each block may represent a function/method in the original program, CBs can be defined by splitting the code into portions of the same size, and CBs can be defined by splitting the original code into portions with a random size. Among the three possibilities, the first one is less effective since it gives a direct correlation between code lines and program functions that can be exploited by an attacker to reverse engineer the program. The other two possibilities can both be applied in an easy way. Once we have the code blocks, we need to properly instrument these blocks in order to make it possible to easily relocate them everywhere in the code section. This means that all control flow instructions (i.e., jump, call, etc.) need to consider the new location where the code is mapped. The Figure 4 shows a typical example. It reports a single code block including two unconditioned jump instructions. We distinguish between two situations. The first situation, represented by the first jump is what we call Intra CB jump, i.e., the program execution jumps to a memory location that is still contained in the same code line. This is the easiest situation.

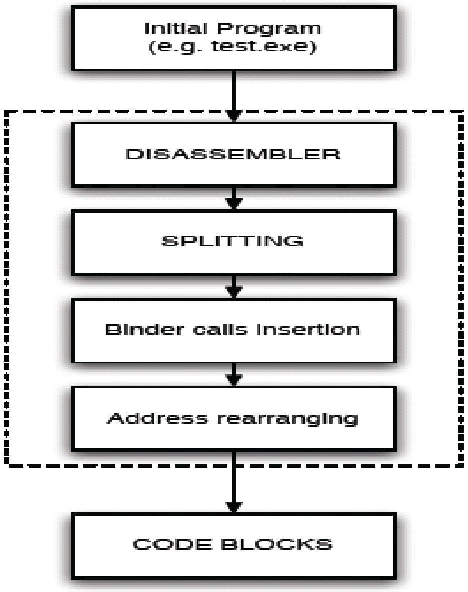


Figure 3. Instrumentation flow

If the target address is expressed as a code displacement (as possible in the Intel IA32 architectures) no modification of the block is required. The second situation is represented by the second jump that we call extra CB jump. It identifies all situations where the program needs to jump to instructions contained in a different code block (the same situation happens in Figure 4 at the end of the execution of the code block).

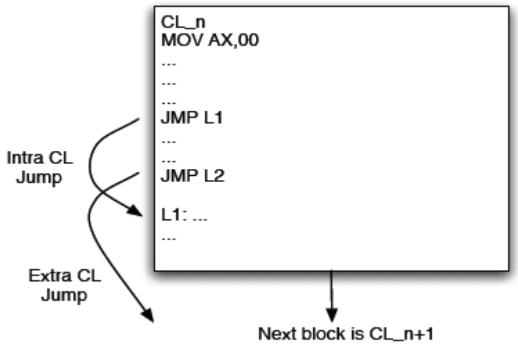


Figure 4. Code Block example

In this case we need to be able to replace this instruction with a call to the binder. Since the target address not always is explained as an immediate value, but can also be contained into a register, this operation must be performed at runtime. Actually, since the trusted node has all the information regarding where the different code blocks are mapped, calculating the target address is a trivial task.

In order to insert the call to the binder we first have to make space for this instruction. We perform this operation by inserting NOP operations (one byte operation) before the target jump to get enough space for the call. In the IA32 architecture the call instruction can be encoded with 5 or 6 bytes, we always consider the worst case.

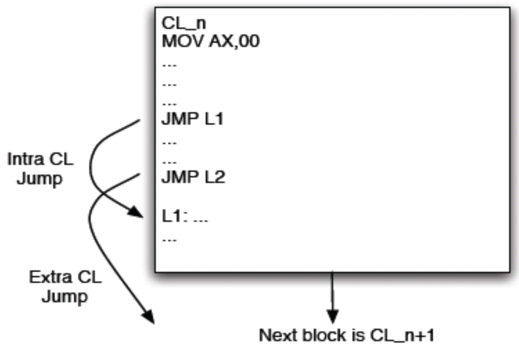


Figure 4. Code Block example

Every time a NOP is inserted all intra CB jumps become inconsistent. Their target address has to be fixed (address rearranging phase) in order to keep the consistency of the block. This operation may present side effects. In particular relative jumps (most of them) can be encoded with addresses on 8, 16 or 32 bits. Injecting NOP operations may lead to the situation where a short jump, i.e., 8 bits address, has to jump to a location outside its maximum range. This in turn requires modifying a short jump into a longer jump. But this operation will lead again to a modification of the code that may in turn require another code inspection to readjust the addresses. This operation must be reiterated until a stable situation is generated.

4. Discussion

An attacker usually tries to reach his goal by disassembling the executable file with tools like IDA [1] and then identifying and removing software protections using debuggers. Our main contribution is exploiting code mobility and continuous dynamic binary code replacement in memory. Indeed, using mobile code blocks extends the control of software providers over released applications beyond the deployment phase. After the release, software is no more at complete mercy of possibly hostile users. In fact, after deployment, the provider retains control of (parts of) the application and is able to apply changes by means of code blocks replacements. Additionally, replacement not only increases the power of software providers, but also lessens the resources of an adversary by capping the attack time. While obfuscation is performed before deployment, our approach takes place during deployment (when the application's memory layout is rebuilt) and even during run-time (when code blocks are stored and/or replaced).

Another important contribution of our approach is providing code splitting at binary level, by using a disassembler during the instrumentation phase before deployment, and the usage of network to continuously transfer a code blocks flow. Moreover in our case the code blocks layout can be customized for different hosts and such information is decided at the server-side and then applied by the Binder. This strongly improves similar approaches such as Binary Fragmentation proposed by Birrer et al. [7]. In this approach fragment locations are chosen at the source code level, thus simplifying the implementation of their metamorphic engine. The problem with this approach is that the binary program layout can be obtained after analyzing the metamorphic engine and it is not customized by the server as in our case. When compared to existing obfuscation techniques, our approach extends prior art in several directions. First, it provides program fragmentation at binary code level and it uses a trusted server to decide a different memory layout for each program instance. Second, and more important, the program is deployed incomplete and binary code blocks are sent by the trusted server to the Binder who executes server commands and actually insert/withdraw code blocks into/from memory. Such quality improves the overall strength of the technique we propose since attackers have limited time resources to check the application's memory layout for each run of the application.

From the attacker viewpoint, disabling code blocks insertion, and thus avoiding correct installation of code blocks, is useless, because the running application would be still incomplete. On the other hand, once the binder is identified among the rest of garbage or useless binary code, the attacker could decompile it, understand its behavior, and replace it with a forged copy. It is unlikely that such a complex attack can be completed manually because the actual layout of blocks is decided by the trusted server and continuously sent at run-time. Moreover, forging the binder does not assure the possibility of mounting useful attacks. The functionality of the binder is limited to the installation of blocks into memory, and no protection tasks are devoted to this element.

To establish how effective our approach is at thwarting reverse-engineers, we applied it to a simple cars-race game with a very simple user interface. The protected program was analyzed using IDA Pro, a popular disassembler/debugger [1]. IDA has difficulty in correctly handling the program: as the program is deployed incomplete, the empty section is filled with random data bytes, and it is not disassembled correctly. The random bytes causes disassembler to shift instruction boundaries (which have variable length in Intel architecture), and displays wrong assembly instructions to the attacker.

Our approach also prevents breakpoints from working as expected in both the free memory area of the program and on the part of it actually filled by code blocks located by the Binder. In the free memory area, if the user places a breakpoint on one of the random data bytes, thinking it is a valid instruction, the breakpoint is never met and does not stop the execution. In addition, placing breakpoints in a code block often cause the program to crash.

The Binder determines the code block to be executed depending on information coming from the trusted server; attackers may try to reconstruct the control flow and the current memory layout by looking at binder behavior but as they need run-time information and debuggers cannot be used properly, reconstructing the control flow is really hard. Every new execution of the program causes the creation of a new session with the server and a new customized memory layout is decided and then disposed by the binder.

The communication with the trusted server can be mediated by software that the attacker controls. Thus the communication can be read and changed by the attacker. Nevertheless, if the attacker breaks or alters the code blocks flow coming from the server he/she cannot use the application.

5. Conclusions

The main contribution of our work is the definition of a new kind of binary obfuscation relying on code mobility and binary code splitting. With our dynamic obfuscation, dynamic analysis is thwarted, as a full binary version of the program is not present in memory at run-time.

Our solution shows that splitting program in code blocks transmitted via network by a trusted server is a suitable and low-cost software protection that can be useful in defending software programs from reverse-engineering. Our protection creates problems for common reverse engineering tools and makes the code comprehension task more difficult for the attacker. By making reverse-engineering more difficult, this technique can help to not disclose proprietary code to competitors. Further research will be devoted to integrate program splitting with other techniques like self-modifying code and remote attestation in order to integrate tamper-detection techniques to improve the level of protection; furthermore we plan to evaluate the increased effort necessary to reverse engineer a binary-obfuscated program (with respect to the effort necessary for a non-obfuscated one) by means of empirical experiments, extending a previous work [8] made on source-code obfuscation.

6. Acknowledgements

This work has been partially funded by the Promising Researcher Award of the University of East London, and by the the European Commission with the project RE-TRUST, which was part of the Future Emerging Technology (FET) program, contract IST-021186. The authors want to thank Prof. Yoram Ofek, the project coordinator, who passed away in the end of 2009; the au¬thors are deeply honored to have worked with him and they will always remember his outstanding contribution to sci¬ence and technology.

7. References

[1] Ida-pro disassembler, http://www.hex-ray. com/idapro/ (Access Date: 12 December, 2010).

[2] Metasm assembly manipulation, http://met asm.cr0.org/ (Access Date: 12 January, 2010).

[3] T2 information security conference challenge. http://www.t2.fi/challenge/ (Access Date: 12 January, 2010).

[4] Trusted computing platform. http://www.trust edcomputing.org/.

[5] D. Aucsmith. Tamper resistant software: An implementa¬tion. In Proceedings of the First International Workshop on Information Hiding, volume 1174 of LNCS, pages 317–333. Springer-Verlag London, UK, 1996.

[6] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sa¬hai, S. Vadhan, and K. Yang. On the (im) possibility of soft¬ware obfuscation. In Crypto 2001, pages 1–18, 2001.

[7] B. D. Birrer, R. A. Raines, R. O. Baldwin, B. E. Mullins, and R. W. Bennington. Program fragmentation as a metamorphic software protection. In 3rd Int. Sympsium on Information Assurance and Security. IAS 2007., pages 369–374, 2007.

[8] M. Ceccato, M. DiPenta, J. Nagra, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella. The effectiveness of source code obfuscation: an experimental assessment. In IEEE International Conference on Program Comprehension (ICPC 2009). IEEE CS Press, 2009.

[9] C. S. Collberg and C. Thomborson. Water- marking, tamper-proofing, and obfuscation -tools for software protection. IEEE Transactions on Software Engineering, 28(8):735– 746, August 2002.

[10] P. Falcarin, R. Scandariato, and M. Baldi. Remote trust with aspect oriented programming. In IEEE Advanced Information and Networking Applications (AINA-06). IEEE, 2006.

[11] J. T. Giffin, M. Christodorescu, and L. Kruger. Strengthening software self-checksu- mming via self-modifying code. In ACM 21st Annual Computer Security Applications Conference, pages 23–32. ACM, 2005.

[12] Y.-J. He, and M.-C. Lee. Towards a secure mutual authentication and key exchange protocol for mobile