International Journal of Engineering and Information Systems (IJEAIS)
ISSN: 2643-640X
Vol. 6 Issue 12, December - 2022, Pages: 82-84

Speculative analysis for an APT detection Treatment of code overlap by static analysis

Mourad M.H Henchiri[1] and Sharyar Wani[2]

[1]KICT, IIUM, Malaysia
mourad@unizwa.edu.om
[2]KICT, IIUM, Malaysia
sharyarwani@iium.edu.my

***Abstracts**: Computer users want to trespass the risks towards their private data. It is required to scan as much code as possible in order to approximate the perfect control flow graph. It is then necessary to overcome on the one hand the incompleteness of a recursive route due to the presence of dynamic jumps and on the other hand the lack of precision of a linear route. One of the challenges is code overlap obfuscation. In this research we would be describing the speculative approach to propose a characterization of binaries using this technique. and this research considers only non-self-modifying programs.*

Keywords: Data at rest, APT, Malware, hybrid analysis, obfuscation, code overlap.

I- INTRODUCTION

Although the code overlap problem is not a recent obfuscation technique and is well documented [1], the disassembly literature often assumes that a byte at a specific address can only be present in only one instruction [19]. This constraint prevents detection of any overlap but allows more precise disassembly on a binary that does not use this protection technique.

From the other side, and in order to analyze self-modifying programs, we propose to divide any execution into a trace and a series of execution snapshots: each snapshot represents a non-self-modifying part of the program. Such implemented solution of this slicing might be an emulation with the Binary Analysis Platform (BAP) and by instrumentation with Pin. The main concern of this research is devoted to the static analysis of the code overlap problem. Besides, and as a perspective, there will be a focus on the analysis of a self-modifying program: we will focus and stress on the concepts discussed, in order to reconstruct the waves corresponding to each level of execution as well as a global control flow graph for the binary studied.

Speculative route: Since the recursive path is less sensitive to very simple obscurations such as the injection of dead code, it is often taken as a starting point in research on static disassembly. After an initial code search with a recursive scan is performed, the remaining bytes can undergo a linear scan that will seek to determine whether they are code or data using heuristics. One of these approaches assesses the likelihood that a string of bytes is actually code by first learning about strings of bytes actually encoding instructions issued during program execution [2, 13]. This sequence of the two routes is called a speculative route.

From the famous proven theories, discussing the speculative routes, and after a first recursive disassembly [3], and when to try to identify the starting addresses of assembler functions using the *push* then *mov* instruction suite, characteristic of compiled functions: they start by stacking the pointer of base stack with push *ebp* to replace it with stack pointer with mov ebp, esp. Also, all the addresses are being identified where a valid instruction is encoded. From these addresses they recursively walk again to an unconditional jump instruction (ret out jmp), believing that the identified code sequence stops on that jump. The risk being great that the paths traversed in this way are not valid, they then eliminate the paths which lead to invalid code or to addresses known to be data.

Examples of overlap

In tElock disassembler: The code in figure 1 is taken from a program protected by tElock and disassembled using a recursive scan from address 0x01006e7a. There is a jmp +1 instruction at the address 0x01006e7d and encoded on the two eb ff bytes, which jumps to the address 0x01006e7d+1 where the dec ecx instruction is present, encoded on ff c9 and which therefore shares the byte ff at address 0x01006e7d+1 with the jmp instruction.

```
Bytes to disassemble: fe 04 0b eb ff c9 7f e6 8b c1
01006e7a    fe  04  0b      inc byte [ ebx+ecx ]
01006e7d    eb      ff      jmp +1
01006e7e    ff      c9      dec ecx
01006e80    7f      e6      jg 01006e68
01006e82    8b      c1      mov eax , ecx
```

Figure 1 - Recursive disassembly of tElock

Disassemblers available: Existing disassemblers, whether using a linear or recursive path, assume that the code cannot overlap and fail to show consistent disassembly otherwise. The recursive disassembly of the example of tElock (figure 1) with IDA Pro (version 6.3) [4] is as follows:

```
01006E7A inc byte ptr [ ebx+ecx ]
01006E7D jmp short near ptr loc_1006E7D+1
; The following bytes have not been disassembled
01006E7F db 0C9h
01006E80 db 7Fh
01006E81 db 0E6h
01006E82 db 8Bh
01006E83 db 0C1h
```

Radare [5] performs the following linear disassembly:

```
01006 e7a f e 04 0b inc byte [ ebx+ecx ]
01006 e7d eb f f jmp 6 e7e
01006 e 7f c9 leave
01006 e80 7 f e6 jg 6 e68
01006 e82 8b c1 mov eax , ecx
```

Neither is able to follow the jump from the jmp instruction: the target of the jump has already been counted as part of another instruction.

Approaches taking into account overlap: The authors of the overlap technique detailed, allowing to encode a hidden code sequence in a sequence [6], propose to detect the protection they expose. The idea is that a long sequence of bytes is unlikely to represent a valid sequence of code. If such a sequence exists, it must be code. So if two long valid strings of code overlap, it is a deliberate obfuscation and one of the two strings contains hidden code. This approach works for the protection they expose but is not applicable to cases of UPX for example because the byte sequences on which instructions overlap are very short and it is plausible that the overlap is accidental and that the overlapping code is not reachable.

II- STATIC ANALYSIS OF CODE OVERLAP

We propose a formalization of the code overlap problem. From a disassembly perspective, a program that has a single instruction that overlaps another can be seen as consisting of a primary disassembly path and a secondary path in which the overlapping instruction fits. Let's take the example of tElock: the segment of bytes eb ff c9 7f e6 can be seen as composed of the two layers of code given in figure 2: there are two layers, the first contains the instructions jmp+1, leave and jg 0x1006e68 and the second contains the instruction dec ecx, overlapping jmp+1. In fact, the eb ff c9 7f e6 byte segment contains exactly the four preceding instructions: there is at most one valid instruction at each address and the last potential instruction, coded on e6, is not valid.

Adresses	0x01006e7d	0x01006e7e	0x01006e7f	0x01006e80	0x01006e81
Bytes	eb	ff	c9	7f	e6
Layer 1	Jmp +1		leave	jg 0x1006e68	
Layer 2	■	dec ecx		■	

Figure 2 - Consistent cutting into layers of tElock extract

For an instruction I we denote the interval of memory addresses on which it is coded C[I]. Formally we define a layer as a set of instructions that do not overlap. Consequently, during disassembly, an attempt is made to perform a coherent division of the instructions included in the control flow graph into different layers. The previous example for tElock is a consistent segregation [7, 8].

III- CONCLUSION

Code overlap is rarely discussed in the literature and, when it is, it is not analyzed as a full-fledged [9, 10, 11] obfuscation technique but rather bypassed by ignoring the range of addresses on which are coded disassembled instructions. We have proposed a notion of code layer, allowing to observe and quantify the use of code overlap by a binary. We propose an algorithm to count these overlaps during recursive disassembly of a binary.

This approach will be taken up and implemented in a related research, in continuation of the current research, to assess the use of overlap by malware.

REFERENCES:

1. Michael Sikorski et Andrew Honig. Practical Malware Analysis : The Hands-On Guide to Dissecting Malicious Software. 1st. San Francisco, CA, USA : No Starch Press, 2012 (pages 24, 27, 57).

2. Nithya Krishnamoorthy, Saumya K. Debray et Keith Fligg. “Static Detection of Disassembly Errors.” WCRE. IEEE Computer Society, 2009, pages 259–268 (page 57).

3. Manish Prasad et Tzi cker Chiueh. “A Binary Rewriting Defense Against Stack based Buffer Overflow Attacks.” USENIX Annual Technical Conference, General Track. USENIX, 3 septembre 2003, pages 211–224 (page 58).

4. Hex-Rays. IDA. https://www.hex-rays.com/products/ida/index.shtml (pages 17, 58, 143).

5. radare. radare, the reverse engineering framework. http://radare.org (pages 16, 59, 143).

6. *Christopher Jämthagen, Patrik Lantz et Martin Hell. “A new instruction overlapping technique for anti-disassembly and obfuscation of x86 binaries”. 2013 (pages 29, 59, 62).*

7. Bishop, C. M. (2006). *Pattern recognition and machine learning*. Springer.

8. Brewer, R. (2014). Advanced persistent threats: minimising the damage. *Network Security*, (pp. 5-9).

9. Brockwell, P. J., & Davis, R. A. (2013). *Time series: theory and methods*. Springer Science & Business Media.

10. Canali, C., Casolari, S., & Lancellotti, R. (2010). A quantitative methodology to identify relevant users in social networks. *IEEE International Workshop on Business Applications of Social Network Analysis (BASNA)*, (pp. 1-8).

11. Intel. Intel 64 and IA-32 Architectures Software Developer’s Manual (Volume 2). http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developerinstruction-set-reference-manual-325383.pdf (pages 12, 27, 44)

12. Chari, S., Habeck, T., Molloy, I., Park, Y., & Teiken, W. (2013). A bigData platform for analytics on access control policies and logs. *Proceedings of the 18th ACM symposium on Access control models and technologies (SACMAT ‘13)*.

13. Data Breaches. (2020, July). Retrieved from World most popular data breaches, Information is beautiful: http://www.informationisbeautiful.net/visualizations/worlds-biggest-data-breaches-hacks.

14. De Vries, J., Hoogstraaten, H., van den Berg, J., & Daskapan, S. (2012). Systems for Detecting Advanced Persistent Threats: A Development Roadmap Using Intelligent Data Analysis. *IEEE International Conference on Cyber Security (CyberSecurity)*, (pp. 54-61).

15. Denning, D. E. (1987). An intrusion-detection model. *Software Engineering, IEEE Transactions on*, (pp. 222-232).

16. Duffield, N. G., & Lo Presti, F. (2009). Multicast inference of packet delay variance at interior network links. *IEEE Computer and Communications Societies*., (pp. 280-285).

17. CAPEC – Common Attack Pattern Enumeration and Classification online Mechanism of Attack at http://capec.mitre.org/data/definitions/1000.html [accessed 1 Jan 2014]

18. Friedberg, I., Skopik, F., Settanni, G., & Fiedler, R. (2015). Combating advanced persistent threats: from network event correlation to incident detection. *Computers & Security*, (pp. 35-57).

19. Christopher Krügel, William K. Robertson, Fredrik Valeur et Giovanni Vigna. “Static Disassembly of Obfuscated Binaries.” USENIX Security Symposium. USENIX, September 18, 2006, pages 255–270.