Subversive-C: How reasonable are the assumptions?

James Cooper
Department of Computer Science
University of Auckland
Email: jcoo092@aucklanduni.ac.nz

***Abstract*—A new form of code re-use attack named Subversive-C, specific to the Objective-C language, and inspired by Counterfeit Object-Oriented Programming, was introduced by Lettner *et al.* in a conference paper published in 2016. In this paper, the authors described an attack that exploits the dynamic messaging system of Objective-C, and which could potentially give an attacker the ability to execute arbitrary code on the attacked computer. The authors then described a defence against that attack, using message authentication codes applied to the Objective-C runtime. In describing both the attack and defence however, the authors make a number of assumptions about the capabilities of the attacker, and other defensive measures in place. This paper will examine some of those assumptions as they pertain to the attack, attempt to assess how reasonable the assumptions are with regards to the attack, with a view to the strength or not of the attack.**

***Index Terms*—Objective-C, Subversive-C, COOP, ASLR, CFI, DEP, XoM, Information leaks, Memory corruption vulnerabilities**

I. Introduction

IN A conference paper [1] published in June 2016, Lettner *et al.* described a code re-use attack inspired by Counterfeit Object-Oriented Programming (COOP) [2] that specifically targets the architecture of the Objective-C language, a language primarily used in applications created for deployment on Apple's OS X and iOS operating systems.[1] Both Lettner's so-called 'Subversive-C' attack as well as COOP are recent variations on Return-Oriented Programming (ROP), a field of techniques used to take control of programs or systems without injecting large new bodies of code. Subversive-C and COOP are interesting in that they rely on underlying features of a specific language (C++ for COOP), which makes these attacks less portable and broadly applicable, but provides avenues of attack that would not necessarily be available in more broadly targeted ROP attacks, which typically work at the Assembly language level. While the use of specific higher-level languages would appear to limit the applicability of the attacks, it is the case however that both C++ and Objective-C are in common use today, meaning that there will be many popular applications installed on users' systems that are vulnerable to these particular attack types.

After describing their Subversive-C attack, Lettner *et al.* then described a method to protect against the attack, one which could be implemented around pre-existing programs without requiring re-compilation or access to the source code. This security measure is based upon using a message authentication code (MAC) with data/instructions that are considered sensitive and in need of securing, with the MAC being computed using a secret key and the contents of the sensitive data. An attacker wishing to modify said data would, in theory, have to know both the secret key, as well as the contents of the data being modified. The defence will not be considered in detail any further in this paper however.

In describing both the attack and the defence, some key assumptions are made. It is not clear if those assumptions are entirely reasonable or relevant, or indeed if they are not potentially contradictory. This paper will critically examine each of the assumptions made, for the paper overall and those made specifically for the attack, and consider whether they are reasonable assumptions that have an impact on the strength of the attack.

A. Relevance of attack

Objective-C has now been superseded by Apple's relatively recent new programming language, Swift. It is likely however that many popular programs that are in common use today, as well as large parts of the main Apple libraries used across their operating systems, are still written partially or perhaps even entirely in Objective-C, so this attack is still relevant. Both languages work on the same runtime, and are interoperable [3, ch. 20], so it is quite plausible that a program implemented in Swift may be vulnerable, at least to some extent, to the same attacks as one written in Objective-C.

B. Technical knowledge

While efforts to explain relevant technical concepts will be made, this paper assumes some knowledge of the terminology of ROP and related fields, for the sake of brevity. The reader is referred to [4] for a good introduction to ROP and associated concepts in general, as well as [5] for a recent overview of the wider field of software attacks.

II. General assumptions for the paper

Lettner *et al.* list some general assumptions made for the paper in Section Three. These assumptions must be examined critically, because the remainder of the paper rests upon them. It is not immediately clear that these assumptions are all valid, nor that they are not potentially contradictory with other assumptions made for the attack (sect. III).

Four general assumptions are made that cover the entire paper [1, s. 3]:

[1]GNU also provides Objective-C support through their GNU Compiler Collection - https://gcc.gnu.org/onlinedocs/gcc/Objective-C.html

1) The attacker can read and write data memory pages, specifically the internal data structures of the Objective-C runtime. Address Space Layout Randomisation is in place however.
2) Data Execution Prevention (DEP) prevents simple code injection attacks
3) The runtime is protected with fine-grained code randomization, and an implementation of Execute-only Memory (XoM) such as Readactor [6] or Execute-No-Read [7]
4) The "C parts of the application and runtime are protected using appropriate mitigations (CFI, randomization, or equivalent defenses)."

These assumptions are asserted to be reasonable, and to match the capabilities of a real-world attacker. None of them are meant to be a 'magic bullet' to stop attackers entirely, as Subversive-C itself demonstrates. They are presumably meant to be a reasonable and realistic list of defensive techniques for a defender to work with, or an attacker to overcome. Nevertheless, they deserve a critical appraisal as they appear in the paper.

A. Read & write access for the attacker

The basic part of this assumption is fairly self-evident. To carry out any sort of code manipulation attack, an attacker will need to be able to read memory to find a location for their purposes, either somewhere they can inject code into, or code they can re-purpose, and then be able to write to inject said code, or modify a return address or in some fashion corrupt the program. On the face of it, having read-write access to memory appears reasonable, if just because without it the attacker likely will be entirely ineffectual, because they cannot corrupt anything.

The proposed attack actually targets individual programs, but the authors state an attacker requires read-write access to the data structures of the Objective-C runtime in order to corrupt them. Subversive-C however targets specific applications and exploits vulnerabilities in those however, so it is not immediately obvious why the attacker needs write access to the runtime specifically.

Address Space Layout Randomisation (ASLR) is also assumed to be in use upon the target system, making such reads and writes more difficult. Davi and Sadeghi [4, p. 56] define ASLR as: "In order to defend against code-reuse attacks, address space layout randomization randomizes the base address of code and data segments per execution run. Hence, the memory location of code that the adversary attempts to use will reside at a random memory location."

ASLR prevents simplistic attempts at code-reuse attacks by situating the relevant code in different areas of memory at each run-time. It is true however that "all ASLR solutions are vulnerable to memory disclosure attacks ... where the adversary gains knowledge of a single runtime address and uses that information to re-enable code-reuse in her playbook once again." [4, p. 56], or, stated alternatively "convential ASLR only changes the base address of the entire code segment and hence a single leaked pointer might uncover all the [attack code's relevant memory] addresses since the relative addresses between them did not change." [7, p. 1344] For example, Snow *et al.* [8] have demonstrated that many ASLR schemes can be overcome effectively with 'just-in-time' code re-use. It is reasonable to expect that some form of ASLR is in place, but it may be the case that it provides only limited protection against the attack.

B. Data Execution Prevention

DEP is a long-standing method to prevent simple code injection attacks. Earlier attacks commonly exploited stack vulnerabilities to place executable code within a program's data, and then ran that code. Normal non-malicious code does not execute from the stack, so all that was needed to prevent these attacks was to prevent code from being run from within the stack. Memory pages are separated into code and data pages, and can only either be written to or executed, but not both, preventing code from being executed from the stack, as well as helping prevent run-time overwrites of the legitimate code in memory. For this reason, DEP is also commonly known as 'Writable XOR Executable', and is supported by all known major modern operating systems and processors. This assumption can be considered entirely reasonable, because of the presence and use of DEP on all major current operating systems.

C. Code randomization and execute-only memory

Lettner *et al.* further assume that the Objective-C "runtime is protected using fine-grained code randomization" as described in [5] (see also [9] for an expanded discussion of the same topics), for example, Marlin [10]. In [5] various mechanisms for code randomization are discussed however, and it is not clear which ones are referred to, so it is difficult to assess this aspect of the assumption further. Note that ASLR can be considered a form of code randomization, because it randomizes the position of code in memory, as well as the relative positions for different code blocks with respect to each other. Many other forms of randomization are possible though.

Different points at which diversity can be introduced are also discussed in [5], including at implementation (code writing) time, compile time, program installation time and program load time. Again, Lettner *et al.* do not state when they they anticipate code randomization to have occurred, but given that the Objective-C runtime is written by Apple, and would already be built and installed on each device by the time the end user starts to run programs using it, the most reasonable assumption is that randomization occurs at program load time, because randomisation performed any earlier could prove to be trivial to bypass for any attacker who is willing to spend time probing for weaknesses before launching their attack.

Along with code randomization, it is also assumed that a form of eXecute only Memory (XoM) such as Readactor [6] or Execute-No-Read (XnR) [7] is in use on the system under attack. The basic principle behind XoM is that code memory cannot be read normally as if it were data memory. Instead, it can only be read by the processor for the purposes of executing said code. This form of protection assists in preventing attackers from locating particular functions in

memory, which stymies attempts to find information leaks (see sect. III-C). Readactor does rely upon features only found in modern processors and releases of operating systems though, while XnR is as-of-yet (to the best of the author's knowledge) not supported by hardware at all.

Crane *et al.* [11] suggest that COOP[2] overcomes XoM, because it targets an element of the program at run-time which is not randomised and protected, specifically the tables holding the memory addresses of C++ virtual functions. The similar nature of Subversive-C to COOP indicates that XoM may be of little help in protecting against the attack. Crane *et al.* present an improved form of XoM, Readactor++, to counteract COOP, but Lettner *et al.* consider it unhelpful in protecting against Objective-C based attacks [1, p. 219] because of the dynamic nature of the messaging system in Objective-C.

It appears to be reasonable to assume that these defences are in place, though it is not entirely clear what forms of code randomization are expected.

D. Control Flow Integrity

Davi and Sadeghi [4, p. 27] define Control Flow Integrity (CFI) as "CFI offers a generic defense against code-reuse attacks by validating the integrity of a program's control-flow based on a predefined [Control Flow Graph] at runtime." A control flow graph (CFG) is, broadly speaking, a graph mapping the valid jump and branch instruction destinations within a program. CFI, in theory, protects against code-reuse attacks by preventing an attacker from redirecting the program's operation to their chosen code for the attack, when that chosen code would not normally be executed in that order according to the CFG. CFI is supported by modern operating systems and compilers.[3] It has been shown however that CFI is not necessarily effective at preventing code-reuse attacks [12]. It also seems possible that, if the CFI schemes used by compilers are ineffective or defective in some way, attackers may know of such weaknesses and potential methods to bypass them. Even partially broken CFI seems likely to be of more use than no CFI at all though.

This assumption is essentially an assumption that both the Objective-C runtime as well as the program to be attacked have both been compiled with CFI functionality enabled in the compiler. It is unclear whether CFI is in use for the Objective-C runtime, but given the critical importance and centrality of the runtime to every Objective-C program, it seems reasonable to assume that, insofar as possible, security measures such as CFI are in place. Apple is surely unlikely not to use such security features where possible.

It further seems reasonable to assume that all recently compiled programs will have taken advantage of CFI support in the compiler, although older programs may well have been compiled before CFI was integrated into compilers. Depending on the exact mechanics of patching, it may or may not be the case that an older program that has been patched would use CFI, since the base program could have been compiled prior to the introduction of CFI. Possibly the more recently patched parts of the program could take advantage of CFI, but older, untouched components do not. Ultimately it would presumably depend on the exact implementation of the CFI scheme and the patching system. All that being said, it might well be the case that recent patches for an older program would have eliminated the memory corruption vulnerabilities (see sect. III-B) that attackers would rely on anyway, and thus the presence of CFI could be a moot point.

Due to the fact that taking advantage of CFI to protect one's programs from having their control flow hijacked is as easy to do as setting a compiler flag, it seems reasonable to assume that at least most current security-concious programs deploy it in some form. Thus, it seems reasonable overall to assume that CFI of some form is in use to protect both the runtime and the target program.

III. Assumptions underlying Subversive-C

Three assumptions are explicitly made for this attack, stated as requirements in [1, s 4.2]. They are:

1) Availability of the AppKit GUI library on the system, as well as its use by the target program
2) Access to a memory corruption vulnerability allowing an attacker to place data in the target process, as well as overwrite a pointer to an Objective-C instance used during execution
3) An information leak sufficient to disclose the memory location of the inserted data and the instance pointer that is overridden with the attacker's counterfeit object

A. Availability of AppKit

AppKit is a library in Mac OS X responsible for providing graphical user interface functionality.[4] The authors require the presence of AppKit as well as its use by the target program because they use the functions present in it for their attack gadgets. It is likely to be present on every up-to-date Apple computer. Furthermore, it is probable that any Mac program with a GUI will make use of the library. Moreover, because a very large number of current programs for Mac will use AppKit for their GUI, it is unlikely that significant changes will be made to the API in the near future. As a result most current programs, and likely many currently being developed, will use the same functions as those relied upon by the attackers. Thus, this assumption is reasonable.

B. Access to a memory corruption vulnerability

Memory Corruption Vulnerabilities (MCVs) [13] are a central aspect of most, if not all, modern code re-use attacks, and "have plagued software written in low-level languages for

[2]It should be noted that, while there are considerable similarities between COOP and Subversive-C, they do differ in the basis of the attacks. Both exploit a specific element of their respective language's operation, namely the C++ vtables for COOP and the dynamic message dispatch functionality of Objective-C for Subversive-C. They could arguably be seen as the equivalent to each other for each language, but the attacks are not directly transferable.

[3]For example, Windows 10 and MSVC++: https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx. It is available to developers using Objective-C on Apple OSes via Clang: http://clang.llvm.org/docs/ControlFlowIntegrity.html

[4]https://developer.apple.com/reference/appkit

more than three decades." [11, p. 243] "This ... covers a large variety of techniques that use programming errors to achieve the same goal: changing the memory contents of the target program." [9, sect. 2.1.1] A quick search on Google Scholar for 'memory corruption vulnerability', limiting the results to only 2015 and 2016, returned over 6,000 results, with 29 of the first 30 being on this topic. It is a much bigger topic than just this paper, and indeed Lettner *et al.* appear to assume that their readers are already familiar with the concept, as they make no attempt to explain it. Fundamentally, it means exploiting a bug that allows the attacker to modify the contents of memory in a way that lets them alter a program's operation to carry out the tasks desired by the attacker.

As such, this assumption is required by the paper to ensure that the attacker can carry out the Subversive-C attack. For Subversive-C, an attacker uses the MCV to place some counterfeit objects into the target program's memory. Those counterfeit objects have been pre-written to suit the attacker's purpose, and will execute a series of illegitimate calls to legitimate functions, in order to achieve an end goal, e.g. executing a call to *system()* on a Linux computer, the example presented by Lettner *et al.*

This assumption, or rather the broad manner in which it is stated, arguably weakens Subversive-C though. A relevant MCV is merely assumed to exist, and to be exploitable by the attacker in a useful fashion. It is likely to be the case that a large proportion of modern popular programs written in Objective-C have MCVs in them, and some of those may potentially prove useful to an attacker, but that is only the start of the attack. It is a laborious and technical process to actually find and take advantage of a vulnerability (see for example [14]).

Subversive-C is however presented only as a proof-of-concept attack, and so it is not necessarily required that it be demonstrated working on a publicly used program. Such a premise lowers the bar that must be overcome for the purposes of the paper, but it does nothing to aid the strength of the attack. An attack that works in ideal circumstances doesn't necessarily work well in any other circumstances. The necessity of a MCV is standard to many papers (e.g. [8], [13]), but Lettner *et al.* merely assume one's existence. This leaves the overall strength of the attack weaker, as arguably the core pillar the efficacy of the attack rests on is hand-waved into place.

Assuming the existence of an MCV would become a smaller weakness in the attack, should the nature of the MCV be described in greater detail. The paper simply states, "... we require a program that contains a memory corruption vulnerability allowing an attacker to place data in the target process as well as overwrite a pointer to an Objective-C instance used during execution." Presumably this was phrased in very broad terms to show that Subversive-C has wide applicability, or perhaps to avoid providing inspiration to would-be attackers reading the article, or maybe simply because Lettner *et al.* did not wish to spend any of their paper's space on what they considered an uncontroversial point. Regardless, by leaving the nature of the MCV entirely undefined beyond describing the overall concept of what it would enable, less evidence is presented that Subversive-C is a realistic attack than might be otherwise.

This is shown to be a gap in the paper's coverage, by the fact that other papers have exploited known vulnerabilities to demonstrate the attack and/or defence, for example [2], [15]. These vulnerabilities likely had already been patched by the time the other papers were published, but by using real-world attacks, extra credence is lent to the idea that they are legitimate threats. It is true that the results of performance tests with real-world programs are shown later in the paper, however those tests do not judge the strength of the attack. They are merely benchmarks for the defence's efficiency.

Considering how critical to the attack a MCV is, it is perhaps unsurprising that one is assumed to be available for Subversive-C. Considering also that MCVs are covered in great detail elsewhere, and the purpose of the paper was to introduce Subversive-C, it is perhaps also unsurprising that the topic was not explored further. A useful MCV is a major assumption however, and simply assuming that one is available without further detail arguably weakens the foundations that Subversive-C rests upon.

It is quite possible that there exist MCVs in any given program, but the ability of any given attacker to find and exploit one that is relevant to their goals is a different, and much less certain, matter. Considering that [1] was about introducing a defence against a potential attack, it is perhaps unsurprising that the attack appears to have some weakness, because the authors' focus was likely on the defence against those potential attacks. A weak attack does a disservice to the presentation of a defence though. A defence that only defends against a weak attack that relies upon ideal circumstances does not necessarily appear to be a particularly strong defence (although note however that the defence proposed does not claim to prevent the existence of MCVs, just prevent their exploitation via the dynamic messaging system of Objective-C). That being said, many programs undergo regular analysis for MCVs by a wide variety of parties, and so it is likely a matter of time before an MCV is located by some party that would be useful to Subversive-C. Therefore, this assumption is reasonable, albeit it is one that ultimately does precious little to reassure the strength of Subversive-C in its own right.

C. Sufficient information leak

Lettner *et al.* state "to reliably bypass ASLR, we ... require an information leak to disclose the position of the data injected and the location of the instance pointer we override with our own counterfeit object." In other words, an information leak is necessary in order for the attacker to use the result of their exploiting an MCV. It seems that an attacker can potentially exploit an MCV without knowing where in memory their attack affected, and thus they must also use the information leak to locate where their attack took place, in order to fully carry it out successfully - if nothing else, they need a base memory address to work from, and randomization may also make locating the AppKit functions used in exemplar attack more difficult. As with MCVs, information leaks are a broad topic with much written on them. Seibert *et al.* [16] present a

number of potential methods for leaking information. Lettner *et al.* give no indication as to what form of information leak they expect to use, simply that there is one they can take advantage of, but Seibert *et al.* demonstrate that one can use a MCV both to corrupt memory in preparation for a ROP-style attack, and for a useful information leak. Most likely it is a fault-analysis attack [16, sect. 3.1].

Furthermore, in the general assumptions Lettner *et al.* make (sect. II-C), they state that along with ASLR they assume that the system is using a form of XoM "that prevents attackers from using information leaks to retrieve the code of the runtime," [1, sect. 3], an assumption they rely on for the defence. They then appear to assume with this information leak assumption that they have the capacity to take advantage of information leaks, which at first glance appears to be an outright contradiction between assumptions. It appears to be the case however that they assume that the *runtime* is protected by XoM, and not the target program. They do not explain why the runtime is protected, and the target program not though. It is not an immediately obvious distinction when first reading, and the paper would have done well to highlight it more, so that readers do not perceive a contradiction that is not there.

If assumption two is held to be reasonable, then [16] suggests that this assumption may well also be reasonable. [1] would have been improved by an explicit mention of this fact though. Such an omission is presumably due to the target audience of [1], being technical experts familiar with ROP and related material.

IV. Conclusion

The four assumptions made for the paper as a whole are reasonable, although more detail on the form of code randomization assumed to protect the program would have been beneficial. The fact that these assumptions largely strengthen the defences in place on the target raise the bar that Subversive-C must get over lends strength to the attack. Two of the three assumptions made specifically for the attack are weaker however, though it is true that if one has one of them, the other may perhaps be available too without much greater effort required. The assumption that the AppKit library is available and in use is reasonable, but the assumptions regarding a memory corruption vulnerability and information leak are less so. They are both necessary for the attack to be carried out, but they are major assumptions that are not explored in the paper. It is likely that there is a memory corruption vulnerability somewhere in most significant programs, but an attacker being able to find and exploit one is less certain.

That being said, [1] is a paper about a defence against a proof-of-concept attack, not a particular attack scenario. The authors may have felt that the assumptions are reasonable without further explanation, as both assumptions are reasonably common in some form in academic papers. Or perhaps they did not wish to spend time on a topic that is covered elsewhere, or maybe because this is a proof-of-concept attack, they felt that justifying the assumptions was a poor use of the space available to them in the paper.

Ultimately, simply assuming a MCV and an information leak into place does little to support the strength of Subversive-C, and could perhaps even be seen to undermine it as a realistic attack method, but there are credible explanations as to why no further detail or explanation was provided. Subversive-C still stands as a potential form of attack on Objective-C programs if the right circumstances coalesce, and developers working with Objective-C would do well to bear the possibility in mind.

References

[1] J. Lettner, B. Kollenda, A. Homescu, P. Larsen, F. Schuster, L. Davi, A.-R. Sadeghi, T. Holz, and M. Franz, "Subversive-c: Abusing and protecting dynamic message dispatch," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Jun 22-24 2016, pp. 209–221. [Online]. Available: https://www.usenix.org/conference/atc16/technical-sessions/presentation/lettner

[2] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A. R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 745–762.

[3] W. Malik, *Learn Swift 2 on the Mac*. Springer Science & Business Media, 2015. [Online]. Available: http://dx.doi.org/10.1007/978-1-4842-1627-9

[4] L. Davi and A.-R. Sadeghi, *Building secure defenses against code-reuse attacks*. Cham: Springer, 2015. [Online]. Available: http://link.springer.com/10.1007/978-3-319-25546-0

[5] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "Sok: Automated software diversity," in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 276–291.

[6] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A. R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical code randomization resilient to memory disclosure," in *2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 763–780.

[7] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny, "You can run but you can't read: Preventing disclosure exploits in executable code," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 1342–1353.

[8] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Security and Privacy (SP), 2013 IEEE Symposium on*, 2013, pp. 574–588.

[9] P. Larsen, S. Brunthaler, L. Davi, A.-R. Sadeghi, and M. Franz, "Automated software diversity," *Synthesis Lectures on Information Security, Privacy, and Trust*, vol. 10, no. 2, pp. 1–88, 12/22; 2016/09 2015. [Online]. Available: http://dx.doi.org/10.2200/S00686ED1V01Y201512SPT014

[10] A. Gupta, J. Habibi, M. S. Kirkpatrick, and E. Bertino, "Marlin: Mitigating code reuse attacks using code randomization," *IEEE Transactions on Dependable and Secure Computing*, vol. 12, no. 3, pp. 326–337, 2015.

[11] S. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, B. D. Sutter, and M. Franz, "It's a trap: table randomization and protection against function-reuse attacks," in *22nd ACM SIGSAC Conference on Computer and Communications Security, Proceedings*. ACM, 2015, pp. 243–255. [Online]. Available: http://dx.doi.org/10.1145/2810103.2813682

[12] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 401–416.

[13] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter, "Breaking the memory secrecy assumption," in *Proceedings of the Second European Workshop on System Security*, ser. EUROSEC '09. New York, NY, USA: ACM, 2009, pp. 1–8. [Online]. Available: http://doi.acm.org.ezproxy.auckland.ac.nz/10.1145/1519144.1519145

[14] E. Perla and M. Oldani, *Stairway to Successful Kernel Exploitation*, ser. A Guide to Kernel Exploitation. Boston: Syngress, 2011, ch. 3, pp. 47–99.

[15] Y. Wang, M. Li, H. Yan, Z. Liu, J. Xue, and C. Hu, "Dynamic binary instrumentation based defense solution against virtual function table hijacking attacks at c++ binary programs," in *2015 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)*, 2015, pp. 430–434.

[16] J. Seibert, H. Okhravi, and E. Söderström, "Information leaks without memory disclosures: Remote side channel attacks on diversified code," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: ACM, 2014, pp. 54–65. [Online]. Available: http://doi.acm.org.ezproxy.auckland.ac.nz/10.1145/2660267.2660309