Introduction to Software Exploitation and Mitigation

Falcon Darkstar Momot, Athabasca University 3172440. 30 November 2014.

Software exploitation by malicious actors is an extremely important area of study, not due to the vulnerability of software, nor due to the escalating skills of attackers. It is important because of the continually increasing reliance of modern society on interconnected information technology systems, and the increased value this presents to an attacker. This learning guide will cover several basic concepts in information security assurance. It will also provide an overview of the ways attackers gain access to systems, and the ways these exploits are mitigated even when software bugs exist. The knowledge so gained will enable the reader to become conversant in information security.

Housekeeping

This learning guide is intended to be self-contained. However, the field of information security assurance is extremely broad in scope. Accordingly, references to external material will be provided in the most persistent manner possible. DOIs or Internet Archive links will be provided where applicable and available. Proprietary information is unavoidable; contact your local academic library.

Objectives

After completion of this learning guide, you will be able to:

- Describe the concept of **conditional security**
- Recognize **security surface** and **probable consequences**
- Create a **threat model**
- Understand non-code-execution threats such as **denial of service**
- Describe pre-mitigation arbitrary code execution attacks such as **stack overflow**
- Describe post-mitigation arbitrary code execution attacks such as **branch-oriented programming**
- Understand the need for countermeasures like **data execution prevention**
- Understand current work around **control flow integrity** and its value

Component 1: Conditional Security

The concept of conditional security lies at the foundation of information assurance. It is simply not possible to secure information assets against all possible threats, both known and unknown; finite resources do not often permit formal verification of software, complete and total quality assurance, and the remediation of all defects. Moreover, as will be addressed in a later unit, an inadequate threat model can render even those efforts moot. An attacker need only succeed once; a defender must defend against all possible attacks.

This asymmetry does not mean that all is lost. Consider the following example. Physical safes are rated according to the expected time it takes to compromise them (Leversage, 2010). If a safe is given a TL-30 rating, for instance, this means that a reasonably skilled burglar with specified tools is expected to be

able to gain access to the safe in no less than 30 minutes. This does not mean we give up hope, and leave our valuables on the countertop instead.

Also, TL-30 safes tend to be more expensive than TL-15 safes, the latter only taking about 15 minutes for the hypothetical average burglar to break into. There are a couple of reasons we might buy a TL-15 rated safe instead of a TL-30 one, even though it is ostensibly less secure: either the contents of the safe are not expensive enough to lose that the more expensive safe is justified, or we know that time constraints mean 15 minutes is more time than the burglar is going to have before a guard comes. This is the basis of conditional security in the physical world.

In the digital world, the concept is somewhat different. There is no telling what an attacker might use to break into our digital safe, but like the burglar's standard tools, we can guess that they will use mostly public exploits and known techniques. Also, instead of periodic guards and watchful neighbours, we have ephemeral utility of information. Cryptography can allow for very long expected times to break in, more comparable to the lifespan of the solar system than to a quarter-hour. Like the burglar, attackers can be expected to compare the value of what they think you have with the costs (and risks) associated with getting it. And like the safe, information security professionals are unable to provide absolute guarantees against compromise.

Conditional security, then, requires that the countermeasures taken to prevent a compromise of information security be chosen in relation to the expected cost of a compromise and the expected value of the compromise to an attacker. The two are often closely linked.

Reading 1.1

Read the Rivest lecture *Conditionally Secure Cryptography* available from the Internet Archive: http://web.archive.org/web/20120616222052/http://web.mit.edu/6.857/OldStuff/Fall97/lectures/lecture4.pdf. Rivest is the R in RSA. Do not worry about understanding Feistel ciphers; concentrate on section 1.1.

Next, peruse the Harvard Tech Report article *Economics and Information Security: a Survey of Recent Analytical, Behavioral, and Empirical Research* available at http://web.archive.org/web/20141130123519/http://ftp.deas.harvard.edu/techreports/tr-03-11.pdf. Concentrate on the idea that an attacker's behaviour has an estimated value associated with it.

Finally, read this tripwire article and attempt to determine whether it has been plagiarized: https://web.archive.org/web/20131203021606/http://www.tripwire.com/state-of-security/featured/conditional-complexity-risk-models

Activity 1.2

Search the Internet for information on conditional security. What are some alternate names for this concept? Why is it not used to market security software and appliances, even though it is used to market safes? What comment does this make on the state of the information security industry?

Activity 1.3

NIST uses the working definition "a measure of the extent to which an entity is threatened by a potential circumstance or event…" for risk. How does this relate to conditional security? Find the rest of the definition from NIST. Who else uses this definition, and in what context? Is it a good definition?

Component 2: Security Surface and Threat Modelling

Security surface is the concept that each component or feature of software adds vulnerabilities and exposure to attackers. Some components, such as DCE-RPC, contribute more to this than others, such as ICMP echo. Everything contributes to this to some extent. Security surface is also often called attack surface.

Security or attack surface can be evaluated by considering factors such as:

- The entry points of input data into the program
- The complexity of input data
- Intended functionality
- Breadth and nature of input sources
- Access services have to each other, and required privileges
- The complexity of functionality
- Known vulnerabilities in services

Analysis of these factors can be conducted in many different ways. Inventorying them is certainly one way. Another method is to model the data flows within the organization. Still another is to inspect each application and determine a series of "channels" for interaction with the service, with discrete functionality and discrete access constraints (Manadhata, 2008).

Limiting security surface is a central pillar of an organization-level approach to information security. It applies still to application security, though to a somewhat lesser extent. It can be handled at several different levels. For example, applications with some security problems can have their security surface reduced in effect by applying isolation.

For evidence of this, consider the concept of the functionality-based access model which underpins SELinux. That model applies security constraints to applications based upon the breadth of system functionality and filesystem contents they are expected to require access to, with a high degree of granularity and independently of their system privileges. They can mitigate many attacks, based on the idea that (for example) a web server should not have direct access to users' password hashes.

Activity 2.1

Read the CVE report for CVE-2014-7985 at http://web.archive.org/web/20141130225300/http://www.cvedetails.com/cve/CVE-2014-7985/.

Consider why such strong language is used to describe the bug's impact. If you were an attacker, what kinds of things could you use this bug for? Can you think of a way to install software on a server running the CRM identified in this vulnerability report?

How would SELinux or a similar functionality-based access model be able to mitigate the attacker's capabilities in this situation? Would it be enough to protect the organization under attack from this vulnerability?

Reading 2.2

Clearly, the correct answer to security in this case is to patch the vulnerability out, by preventing directory traversal in the involved paths, and preventing directory backtracking in particular.

Unfortunately, this is rarely feasible for IT operations, and must usually be addressed by software vendors on their own schedule. This puts IT into a problem: for any given software, the software probably has disclosed and undisclosed vulnerabilities, and yet, you have to run at least some software. This is why sandboxing and isolation techniques (including virtualization) are as popular as they are.

Locate the article introducing a sandboxing framework called Alcatraz, using http://dx.doi.org/10.1145/1455526.1455527. Consider how Alcatraz impacts the security surface of an application in an enterprise context.

Activity 2.3

Read through the Open Web Application Security Project cheat sheet for determining an application's attack surface. The URL can be found in the references (Bird & Manico, 2014). The presented framework is designed for web applications, but has concepts that can be applied very broadly. Consider a web application you have experience with, and evaluate it using that framework.

Reading 2.4

Firewalls are another popular method of reducing security surface. This method works primarily at the network level, rather than at the application level; the idea of viewing an enterprise network as a single integrated system for the purposes of attack is a useful one when considering the idea of security surface.

Consider Microsoft's documentation on reducing the attack surface of their Forefront Threat Management Gateway product: https://web.archive.org/web/20130517202202/http://technet.microsoft.com/en-us/library/cc995072.aspx

The strategies Microsoft presents for reducing the attack surface of that product generally involve removing functionality that is not required. The reasoning here is that complexity, and thus vulnerability, increases with functionality. Accordingly, unused features should be disabled where possible: if a vulnerability in the TMG's VPN access features would result in a compromise, but they were not enabled, the compromise could not have happened and the network is safe from that particular problem.

This is a very close functional equivalent to the role of a firewall in an organization. For example, consider the Microsoft Security Bulletin for an exploitable RPC bug: https://technet.microsoft.com/library/security/ms08-067

Activity 2.5

Read through the security bulletin. These bulletins are often released alongside security updates, and are targeted at information security professionals who wish to secure their network, manage the patching process, and determine if they have been compromised; this is an extremely good practice which will be considered later.

What can you tell about the vulnerability in the security bulletin?

The bulletin directs users to use firewalls to restrict access to two ports, used for RPC: the NetBIOS over TCP port, and the SMB over TCP port. Why might Microsoft recommend this even though the

vulnerability can be patched? What is the cost-benefit analysis of disabling Internet access to those ports? How about limiting or segmenting Intranet access to those ports?

Can you model a hypothetical scenario where disabling Intranet access to RPC ports makes sense? Explain such a scenario, and what access would be blocked.

Reading 2.6

The process of analyzing the last scenario in Activity 2.5 created a very simple threat model, as well as demonstrating that threats can come from within an organization, as well as without it.

Read the threat model for Active Directory presented at https://web.archive.org/web/20141201012639/http://kar.kent.ac.uk/14086/1/THREAT_MODELLING_FOR_ACTIVE.pdf. While reading it, consider in what ways it is incomplete.

A threat model is only as useful as its completeness. By now, it should be apparent that threat model completeness is a limit which can be approached yet not totally reached; the scope of possible vulnerabilities and avenues to attack is very large.

Drawing on attack surface concepts from earlier in the unit can help to frame the threat model. For instance, Active Directory permits an organization to store detailed personal information and pictures for every user in an enterprise. Custom attributes could be added for even more information. But, if an organization does not use this functionality and the data is not present, there is no reason to include it in the threat model and little reason to mitigate it. In light of this, is the fact that the threat model is vague on the information disclosure point really a weakness? Probably not, but it does demonstrate that a threat model is much stronger when it is developed with an understanding of how software is used.

Read about the STRIDE method for threat modelling at the OWASP: https://web.archive.org/web/20140913213910/https://www.owasp.org/index.php/Threat_Risk_Modeling#STRIDE

Activity 2.7

Pick any software you have experience using, preferably in an enterprise setting. Create a STRIDE threat model for it. What do you think are the most pertinent threats based on this model? How would you mitigate them? Do you need the vendor's help to mitigate them?

Use the DREAD method, later in the same OWASP document, to evaluate the most pertinent threat you selected. What is a shortcoming of DREAD? Is it an effective tool?

Component 3: Denial of Service Attacks

A great deal of the effort expended in information security is around preventing unauthorized access, remote code execution, and viruses. Denial of service is another attack type which has become increasingly important in recent years, both for financially (Zeifman, 2014) and ideologically (Mutton, 2010) motivated attackers. These attacks are fairly difficult to guard against; attackers have several asymmetric advantages to choose from. Software vulnerabilities can be exploited as mentioned in the analysis of Active Directory from the previous component, but it is also possible to use software vulnerabilities in other software or simply amass a large amount of bandwidth to consume network

capacity and request handling capacity. These attacks can cause serious problems for businesses, by causing reputational damage and business interruption, and in this sense are very similar to other attack types.

A classic method of denial-of-service is the DDoS attack, where a large number of compromised hosts are used to flood a smaller number of (higher-capacity) hosts with traffic. Other methods involve using a software vulnerability that knocks a service offline, or causing other hosts to send traffic to the target.

Reading 3.1

One very popular method for executing denial-of-service attacks is the DNS reflection attack, where a number of DNS servers are used to flood a host. This is an amplification-type DoS attack, which leverages the very high bandwidth and large potential DNS response size compared to query size to send more traffic to a host than an attacker could otherwise manage.

Read this paper on the attack and its mitigation strategies to better understand the mechanics:

https://web.archive.org/web/20141201020219/http://work.delaat.net/rp/2012-2013/p29/report.pdf

Consider whether the increased overhead of connection-oriented DNS over TCP is worthwhile if it makes this attack more difficult to execute.

Reading 3.2

For a broader overview of DDoS attacks from an IT perspective, read this canonical Server Fault Q&A:

http://serverfault.com/q/531941/126699

While reading it, focus on a network service that you have provided in the past, or if you haven't, focus on hypothetically hosting a website.

Then, read this hosting provider's documentation for individuals whose IP address has been null-routed as part of a DDoS mitigation strategy:

https://web.archive.org/web/20141201021708/http://www.memset.com/docs/additional-information/ddos-and-null-routing/

Activity 3.3

Evaluate the methods provided in the Server Fault Q&A. What is the most effective solution to a traffic-based DDoS? Explain one situation in which it would be cost-effective, and another situation in which it would not.

Component 4: Arbitrary Code Execution Attacks

You have already seen how not all attacks relate to remote code execution: the information disclosure aspects of Active Directory and the possibility for DNS to be used to execute denial-of-service attacks show two very important (but far from exhaustive) threats that involve no arbitrary code execution at all.

The more classic approach to software exploitation, however, is getting arbitrary code to run on a target. Vulnerabilities which allow this are sometimes called "remotes", and ones that are not publicly known are called "0-day" (i.e. the patch has been available for 0 days). The goal of this section is to

somewhat demystify computer network exploitation and the tools available to hackers; it will barely be able to scratch the surface of this topic.

There are many techniques that can be used to gain remote code execution. This learning guide will concentrate on only two. Stack overflow exploits are covered because of their foundational nature. Return-oriented programming (a subclass of branch-oriented programming) is covered because it is a modern technique that gives an effective example of mitigation bypassing.

Reading 4.1

Aleph One's seminal phrack paper explaining return address overwrites created by exploiting stack buffer overflow bugs is considered one of the most important works in computer security. It was published in 1996. Read and understand it:
http://web.archive.org/web/20140909012818/http://phrack.org/issues/49/14.html

It shouldn't be difficult to see how the same concept applies to heap buffers and vtable pointers, if you are familiar with compiled C++ code. Many instances of this type of bug continue to exist to this day.

Activity 4.2

This type of attack is very easy to conduct, relatively speaking, but computer network exploitation is a different mindset and can be very difficult in the beginning. Go through this exercise:

https://web.archive.org/web/20140405143139/http://exploit-exercises.com/protostar/stack4

The exercise provides a vulnerable program which must be exploited by overwriting a return address, in the manner explained by Aleph One. In this case, no shellcode (code an attacker introduces into a process, typically to get a shell on the exploited computer) is actually required; the goal is only to redirect execution.

For an overview of this type of technique in general and a tutorial on writing shellcode, read the book "Hacking: The Art of Exploitation" by Jon Erickson. The latest edition as of this writing is the second.

Reading 4.3

The obvious way to prevent this type of attack is to check buffer bounds. This has been a standard thing to teach programmers who are new to writing native code; still others elect to avoid native code and only write managed code in an attempt to avoid this problem. As these remote code execution techniques are the vast bulk of exploit techniques mitigated by writing managed code (also consider use-after-free attacks and branch-oriented programming), it is safe to conclude that they are largely responsible for the idea that native code is less secure than managed code. With some knowledge of threat modelling and varied attack techniques and goals, it is not hard to deconstruct the idea that managed code is intrinsically secure.

After some time, circumstances required and computing power allowed for automatic mitigation of these techniques through the prevention of data execution. Excluding programs from executing data they can also write to is one popular strategy for preventing arbitrary code injection; this technique has many names, such as Data Execution Prevention, Advanced Virus Protection, NX, and W^X.

For a short explanation of this mitigation in OpenBSD, read Theo de Raadt's mailing list post on the topic: http://marc.info/?l=openbsd-misc&m=105056000801065

This prevents attackers from injecting code into a process and then calling it. But, this can be bypassed too.

Reading 4.4

A favourite technique of attackers is to use functionality present in systems in ways that the creators of those systems did not intend. Without the ability to inject arbitrary code into processes, it has been necessary for attackers to use code already present in those processes to do arbitrary things. This is far from an impossible task.

Imagine a program which has a vulnerable buffer that allows the return address of the current function to be overwritten, just as identified by Aleph One. In a program where Data Execution Prevention or similar has been applied, it will not be possible to execute Aleph One's attack per se. However, the attacker can still return to any arbitrary location within a program.

A series of return addresses, rather than a single return address, can also be used to execute complex programs. An attacker can do this by seeding the stack with many return addresses, and then allowing execution to proceed to the end of the current function; the return address will execute a short series of instructions gathered from the tail end of a function (this is called a gadget), and then return to the next fake return address. Programs of nearly limitless complexity can be composed out of these sequences, especially in x86 where bytecode can have very different meanings when execution is directed into the middle of an instruction.

For a description of code reuse attacks in general, which comprise both types of branch-oriented programming (both return-oriented and jump-oriented programming), read the journal article at http://dx.doi.org/10.1109/TC.2012.269 (if your institution has access to IEEEXplore, you should be able to access the article through your library's proxy).

Component 5: Control Flow Integrity

A great deal of academic work around exploit prevention in native code has been in the area of control flow integrity (CFI). The CFI idea is that program control flow can be expressed as a directed graph giving the flow of control between different blocks of a program, and that any deviation from this flow represents positive evidence that the running program has been compromised. CFI clearly mitigates both code injection and branch-oriented programming exploits, at least in theory. Its drawbacks include high performance cost due to high complexity, general tractability problems with static analysis, and high invasiveness. However, it is a promising approach, and much work has been done around it.

Reading 5.1

An interesting parallel solution to the code reuse attack problem is presented by Hiser et. al.: http://dx.doi.org/10.1109/SP.2012.39. Read their work. Consider its inherent inefficiencies, and whether it protects effectively against code reuse attacks. Consider whether it protects against code injection attacks.

This work also talks about information leaks in brief, by way of introducing a mitigation purporting to make them useless for computer network exploitation. An information leak is the mechanism an attacker can use to bypass techniques such as ASLR, which also attempt to block code reuse attacks. Hiser et. al. is an extension of the ASLR idea.

Activity 5.2

Locate the paper *DROP: Detecting Return-Oriented Programming Malicious Code* at http://dx.doi.org/10.1007/978-3-642-10772-6_13. Evaluate the technique presented by Chen et. al. Discuss the validity of the fingerprinting technique used and the claims made by the authors.

The DROP technique offers a very good improvement over graph-based approaches to control flow integrity, by detecting the technique based on its immutable differences from normal code. However, the technique is not able to detect ret-to-libc attacks (explained in brief in that paper) which are *not* ROP attacks.

Is there some way an attacker can return into libc or another standard library without using gadgets, and instead using whole functions? How are library functions similar to system calls from the perspective of an attacker? Is there a way to detect ret-to-libc attacks?

Reading 5.3

Read the paper *A Method for Detecting Obfuscated Calls in Malicious Binaries* at http://dx.doi.org/10.1109/TSE.2005.120 (very similar work has been published by the same authors many times). This technique leverages graph theory and an abstract model of program execution to determine whether or not a program is following its intended control flow or whether execution has been hijacked; this represents a traditional approach to CFI.

References

Bird, J., & Manico, J. (2014). *Attack Surface Analysis Cheat Sheet* [wiki page]. Retrieved 30 November 2014 from https://web.archive.org/web/20141130231004/https://www.owasp.org/index.php?title=Attack_Surface_Analysis_Cheat_Sheet

Chen, P., Xiao, H., Shen, X., Yin, X., Mao, B., & Xie, L. (2009). DROP: Detecting return-oriented programming malicious code. *5th International Conference, ICISS 2009*, 163-177. doi:10.1007/978-3-642-10772-6_13

Hiser, J., Nguyen-Truong, A., Co, M., Hall, M., & Davidson, J. W. (2012). ILR: Where'd my gadgets go? *2012 IEEE Symposium on Security and Privacy*, 571-585. doi:10.1109/SP.2012.39

Kayaalp, M., Ozsoy, M., Ghazaleh, N. A., & Ponomarev, D. (2014). Efficiently Securing Systems from Code Reuse Attacks. *IEEE Transactions on Computers, 63*(5), 1144-1156. doi:10.1109/TC.2012.269

Lakhotia, A., Kumar, E. U., & Venable, M. (2005). A method for detecting obfuscated calls in malicious binaries. *IEEE Transactions on Software Engineering, 31*(11), 955-968. doi:10.1109/TSE.2005.120

Leversage, D. J., & James, E. (2008). Estimating a system's mean time-to-compromise. *Security & Privacy, IEEE, 6*(1), 52-60. doi:10.1109/MSP.2008.9

Liang, Z., Sin, W., Venkatakrishnan, V. N., & Sekar, R. (2009). Alcatraz: An isolated environment for experimenting with untrusted software. *ACM Transactions on Information and System Security, 12*(3), 14. doi:10.1145/1455526.1455527

Manadhata, P. (2008). *An Attack Surface Metric* [doctoral thesis]. Retrieved 30 November 2014 from https://web.archive.org/web/20131102005500/http://reports-archive.adm.cs.cmu.edu/anon/2008/CMU-CS-08-152.pdf

Microsoft (2008). *Vulnerability in Server Service Could Allow Remote Code Execution (958644)*. Retrieved 30 November 2014 from https://web.archive.org/web/20141104170342/https://technet.microsoft.com/library/security/ms08-067

Moore, T., & Anderson, R. (2011). *Economics and Internet Security: a Survey of Recent Analytical, Empirical, and Behavioral Research.* Retrieved 30 November 2014 from http://web.archive.org/web/20141130123519/http://ftp.deas.harvard.edu/techreports/tr-03-11.pdf

Mutton, P. (2010). *MasterCard Attacked by Voluntary Botnet After WikiLeaks Decision.* Retrieved 30 November 2014 from https://web.archive.org/web/20141005214321/http://news.netcraft.com/archives/2010/12/08/mastercard-attacked-by-voluntary-botnet-after-wikileaks-decision.html

National Institute of Standards and Technology (2012). *Guide for Conducting Risk Assessments*. Retrieved 30 November 2014 from https://web.archive.org/web/20140716033244/http://csrc.nist.gov/publications/nistpubs/800-30-rev1/sp800_30_r1.pdf

One, A. (1996). Smashing the stack for fun and profit. *Phrack, 49*. Retrieved 30 November 2014 from http://web.archive.org/web/20140909012818/http://phrack.org/issues/49/14.html

Peisert, S., Bishop, M., & Marzullo, K. (2010). *What do Firewalls Protect? An Empirical Study of Firewalls, Vulnerabilities, and Attacks*. Retrieved 30 November 2014 from https://web.archive.org/web/20141201011938/http://nob.cs.ucdavis.edu/bishop/notes/2010-cse-8/2010-cse-8.pdf

Rivest, R., & Goldberg, M. (1997). *Conditionally Secure Cryptography* [lecture notes]. Retrieved 30 November 2014 from http://web.archive.org/web/20120616222052/http://web.mit.edu/6.857/OldStuff/Fall97/lectures/lecture4.pdf

Rozekrans, T., & Koning, J. (2013). *Defensing against DNS Reflection Amplification Attacks.* Retrieved 30 November 2014 from https://web.archive.org/web/20141201020219/http://work.delaat.net/rp/2012-2013/p29/report.pdf

Schreuders, Z. C., Payne, C., & McGill, T. (2013). The functionality-based application confinement model. *International Journal of Information Security, 12*(5), 393-422. doi:10.1007/s10207-013-0199-4

Vulnerability Details: CVE-2014-7985 (2014) [website]. Retrieved November 30, 2014 from http://web.archive.org/web/20141130225300/http://www.cvedetails.com/cve/CVE-2014-7985/

Wen, Y., Lee, J., Liu, Z., Zheng, Q., Shi, W., Xu, S., & Suh, T. (2013). Multi-processor architectural support for protecting virtual machine privacy in untrusted cloud environment. *Proceedings of the ACM International Conference on Computing Frontiers*, 25. doi:10.1145/2482767.2482799

Zeifman, I. (2014). *Ransom DDoS: Criminal Masterminds or Kids with Nukes?* [blog article]. Retrieved 30 November 2014 from https://web.archive.org/web/20140702210637/http://www.incapsula.com/blog/ransom-and-blackmail-ddos.html