THE THREE LEVELS OF EXPLOIT TESTING

Richard Ford & Marco Carvalho
Florida Institute of Technology, USA

Email {rford, mcarvalho}@fit.edu

ABSTRACT

Many different client-side security products claim to provide protection against exploits from known and unknown vulnerabilities. However, to date, scientific tests of exploit detection solutions have been lacking, and those that have been conducted are of limited utility.

In this paper, we explore three different types of tests that could be used to measure the efficacy of products that claim to provide protection, and discuss the properties that these tests would actually measure. We argue that each test measures a different property of the protection provided, and that none measure the functionality most purchasers actually care about. We then illustrate how existing tests can provide misleading information about the actual efficacy of measured products.

At the simplest level, exploit detection can be measured by testing machines using known-vulnerable versions of software against exploits taken from Metasploit. However, this test fails to distinguish between products that detect known exploits targeting known vulnerabilities and products that simply detect the presence of known exploit code. At the next level, tests could be conducted using new exploits for known vulnerabilities. These could be created by building new exploits for well-documented vulnerabilities; however, this test does not discriminate between products that rely on prior knowledge of the exploit conditions and those which do not. Finally, we propose a new test that we believe raises the bar in exploit detection testing: the measurement of new exploits targeting new vulnerabilities. We provide a methodology that allows testers to leverage new vulnerabilities ethically and efficiently, and consider how an unscrupulous vendor might attempt to 'game' this new methodology. We demonstrate how these tests can be used in a cost-effective manner, and show how this test can further be enhanced with minimal effort, by adding techniques that allow testers to discriminate between post-exploitation behavioural detection and actual detection of the exploit.

INTRODUCTION

As endpoint security solutions become ever more encompassing in terms of functionality, testers face significant challenges in designing and executing tests that adequately measure product efficacy. This is especially true for products that claim to provide 'generic' solutions to entire classes of attack. For example, it has proven difficult to measure the utility of generic malware detection without actually creating new malware to test against – something that is typically not done, for both ethical and practical reasons.

This problem will continue to evolve in both its complexity and tractability as new technologies are brought to bear against attackers. In this paper, we offer a solution to just one of these new challenges: how to test exploit detection solutions.

At first glance, testing an exploit detection system seems easy: one simply installs the countermeasure on a vulnerable system and exploits it. Indeed, as we shall see, such an approach does measure the ability of the product to detect known exploits of known vulnerabilities, but it may not be able to measure reliably how well the product will handle new exploits or, even more importantly, how well it will handle new vulnerabilities.

Before moving on, it is worth us defining our terms. When we describe a system, we distinguish between *vulnerability* and *exploit.* The vulnerability represents the underlying security weakness. An example of a vulnerability might be reading a user-provided string into an unchecked buffer. In contrast, an *exploit* is the active exploitation of an existing vulnerability. For any particular vulnerability, there may exist many possible exploits. For example, for a buffer overrun, there are a wealth of different strings that could be used as exploits, all with different payloads and structure.

A second definitional aspect is what we mean by 'a measure' and 'a metric'. Typically, Stevens [1] would describe a measure as something that we can simply quantify, such as height or weight. These simple measures are direct and straightforward. When it comes to more complex properties, such as product effectiveness, we will generally talk of metrics – indirect measures that give us some insight into the more complex property we wish to assess.

For exploit detection, then, we will be searching for a metric that gives insight into product efficacy with respect to exploit detection. In general, such detection can be accomplished at many different levels. We argue that the real functionality of interest is the ability of a service provider to detect the exploitation of an unknown vulnerability. Ideally, the technology should detect the exploit itself, not the actions that the attacker takes post-exploitation. We make this argument based on the many different motives of attackers, making post-exploitation behaviour highly variable.

As an outline, the paper considers the following issues: we begin with a quick overview of the state of the art in exploit detection and review some current testing methodologies, highlighting their weaknesses. We then describe an approach to exploit detection testing that relies upon the creation of artificial (but fully exploitable) vulnerabilities. This approach allows us to quickly and easily test the claims made for the capabilities of exploit detection software. Finally, we consider how this work might be put into practice, and how tests of this nature help shape the capabilities of developing products.

EXAMPLES OF EXPLOIT TESTS

One of the suggestions often seen for testing IDS/IPS solutions is Metasploit. This project provides information about security vulnerabilities and is aimed at helping to generate IDS signatures and deploy exploits. The thought is that a protected system could be exposed to attacks via a representative threat vector.

Metasploit is a fairly popular tool and has a large number of both commercial and open-source front ends to increase its usability. Thus, its pervasiveness, well-understood features and functionality, and its (apparent) suitability to the task of testing exploit detection software make it a frequently suggested solution. However, as we will see below, the use of Metasploit for this purpose is not without drawbacks.

Another possible approach was implemented by *MRG Effitas* [2]. In this case, testers carried out a test that was something of a hybrid between known vulnerability testing and real-world testing. In summary, the tests worked as follows:

1. A *Windows 7* machine was installed, and the security of the OS was weakened by turning off UACX, *Microsoft Defender* and *Internet Explorer Smartscreen*.
2. Vulnerable web-facing software (Java 1.7.0 etc.) was installed.
3. *Windows Update* was enabled, but only for patches that did not affect *Internet Explorer*.
4. Protection products were installed on snapshots of the vulnerable system.
5. Malicious URLs were crawled, while monitoring software looked for new processes (malware) to start.
6. After successful exploitation, the machine was reset and the image was reset.

As noted above, this tests against known vulnerabilities, but it is unclear whether the exploits are known or not.

Finally, another option is a 'clinical trial model' real-world test, as espoused by Somayaji *et al.* [3], and implemented by Levesque *et al.* [4]. In such an approach, the trajectories of real user machines are monitored and the impact of treatments (e.g. patching, different anti-malware software) measured. In some ways, this represents the 'perfect' test as it measures the actual efficacy of the 'as-deployed' solution, but there are still drawbacks, as we discuss below.

WEAKNESSES

When we consider the utility of a detection technique it is important that there is a very clear understanding of the use case that we are attempting to measure. This is perhaps best illustrated by a better-known example: the value of testing anti-malware static detection in an environment where cloud-based services have been turned off.

Typically, tests of static scanning with no cloud services are criticized by vendors; they argue that this is not a valid test, as it measures a feature that they claim is irrelevant to users. However, their objections miss an important question: is there a real-world *use case* that this test measures? The answer, after some thought, is yes. There are many example environments that are air-gapped from the Internet for either security or operational reasons. An embedded system in a highly sensitive application (such as the flight deck of a military aircraft) may need to be scanned for malware at certain points in time. These embedded systems that are disallowed from being connected to the Internet make such a test meaningful to a subset of the population. Conversely, for a typical home user, there is indeed little to be learned from examining test results of cloud-disabled solutions. The lesson here is that for every test scenario we must be clear about the use case that is being tested.

In the case of exploit detection, it is typically not the exploitation of known exploits that we are concerned with. The use case is *not* that we wish to detect a known exploit at runtime; instead, we would very much like to know that the system is protected from 0-day threats (the reasoning being that most known threats can be patched fairly quickly). There is a limited use case for known exploit detection, but this functionality is typically not as important as 0-day protection, and would be subsumed by accurate 0-day protection.

Thus, the real issue here is that none of the techniques outlined above are actually a measure of the behaviour we care most about: none of them tell us whether a product under test can detect exploitation generically, or whether it simply detects exploits and/or vulnerabilities that are already known about. We argue that a good test of exploit detection should differentiate between these two cases.

CAN WE DO BETTER?

As touched upon above, there are different levels of protection that can be provided by exploit detection systems. In order of increasing utility these are:

1. Detection of a known exploit of a known vulnerability.
2. Detection of an unknown exploit of a known vulnerability.
3. Detection of an unknown exploit of an unknown vulnerability:
   a. Based on post-exploit behaviour.
   b. Based on detection of the act of exploitation itself.

Technique	Known vulnerability	Known exploit	Strength	Weakness
Metasploit with stock exploits	Yes	Yes	Cost effective	Does not show that the product actually detects exploitation; can detect simple exploit signature
Metasploit with custom exploits	Yes	No	Test determines product ability to detect exploits beyond simple exploit signature	Does not demonstrate ability to detect 0-day vulnerabilities
Real world	Maybe	Maybe	Gives real-world efficacy of protection, including impact of user acceptance etc.	Many clients needed

Table 1: Exploitation detection measurement schemes and their strengths and weaknesses.

As outlined above, it is relatively easy to test the system at levels 1 and 2; for example, use of a known-vulnerable system and Metasploit will test both of these countermeasures easily. However, testing at levels 3a and 3b – the levels we care most about – seems very difficult.

One way of testing at these higher levels is to use automated tools to find new vulnerabilities in software, develop exploits for them, and then use these 0-day attacks to test software. However, there are two problems with this approach: finding exploits is neither easy nor cheap, and providing verifiable and repeatable test results will require the dissemination of new vulnerabilities for real software. Realistically, finding one's own 0-day exploits for exploit detection testing is not scalable. While it is conceivable that one could 'freeze' a security product, disconnecting it from the cloud, and use exploits that are new to the version stored, this approach is sub-optimal, and would unfairly penalize those products that use network connectivity to enhance detection. While it is not obvious that exploit detection software would make significant use of the cloud, tests should not presuppose how protection is provided.

A second issue with finding one's own vulnerabilities is that the vulnerability should be disclosed to the developer of the vulnerable software following responsible disclosure practices… and good testing dictates that tests are transparent. There is therefore a rather difficult tension between the needs of the test and the needs of the vulnerable software developer. Ultimately, finding one's own 0-days for the purposes of software testing is impractical.

Another approach to testing at level 3 is to deploy systems in a real-world setting and see how and when they are broken into. There are two fairly significant problems with this particular approach. First, the rate of new attacks is low and the variance between machines is likely very high. Thus, a 'clinical trial model' [3] would require an extremely large number of machines in order to produce statistically meaningful results. Second, and perhaps most importantly, even if a sufficient number of participants could be recruited, it would (by definition) be impossible to know about exploits that were *not* detected.

To this end we propose that a more fruitful approach is to write vulnerabilities deliberately into custom copies of open-source software, creating a new package with a new (and known) vulnerability. Thus, there is no ethical concern about distributing a new vulnerability, as it is not usable elsewhere. In addition, there is no question mark regarding undetected exploits, as the tester has complete control of the test.

As such, our proposal is simple: deliberately create new vulnerabilities in a package such as *Apache*, and then exploit them in a variety of different ways.

INJECTABLE VULNERABILITIES

The real benefits of our approach become apparent when we consider the large number of different vulnerability types that exist. There have been several attempts to document different vulnerability root causes and to generate a taxonomy (see [5, 6] for an overview and list of references), but any list inevitably becomes out of date quickly as new exploit techniques are developed frequently. Nevertheless, we can choose the type and scope of vulnerability we choose to inject into the source code. Vulnerability databases, such as the CVE list and National Vulnerability Database (see: http://cve.mitre.org/), can also provide solid starting points for determining what kinds of vulnerability are common. As such, the relative importance and frequency of vulnerability types can be gauged in order to build a test set of exploits and vulnerabilities.

This ability to choose the type of vulnerability affords the tester significant control over the way in which a product is tested. For example, when we consider a stack-based buffer overrun in a network server, there are many different ways in which the intrusion could be detected.

A few of the possible defence approaches are listed below:

- A protocol analysis could be carried out, and the anomaly in the network request could reveal the attack.
- The protection system could note the destruction of the return IP address on the stack (for example, using stack cookies).
- The anomalous control flow change could be detected.
- The behaviour of the payload (e.g. downloading a trojan from a known-bad IP address) could reveal the attack.

As can be seen, the 'same' protection can be provided in many different ways. Note that, in fact, the word 'same' here is intentionally in quotation marks because in truth these protection schemes are *not* equivalent. Depending on the use case, one may be vastly preferable to another, and by controlling the vulnerability (and by having full knowledge of the code that surrounds it), the tester can explore this space with relative ease. Bespoke vulnerability creation (as opposed to exploit creation) allows this space to be explored methodically and controllably.

WHAT A TEST WOULD LOOK LIKE (AND WHAT IT MIGHT TELL US)

The best illustration of the value of our testing technique is a worked example of how an exploit test using our techniques might look. To that end, we perform a *Gedanken* of a fictional product test that attempts to determine the efficacy of two products that use different protection techniques.

As a first step, the use case to be tested must be determined. In this instance, the tester wishes to answer the most general detection question: which product provides the most effective protection from exploitation of 0-day attacks? With this overall question in mind, the tester can set about the task of creating a test.

Next, a set of vulnerabilities should be selected from any appropriate taxonomy. For example, the tester may choose to test buffer overruns, TOCTTOU vulnerabilities and data disclosure, justifying this choice by citing issues in the NVD. In each case, an open-source software package would be modified such that it was vulnerable to one or more of these weaknesses. With exact knowledge of the source code and the vulnerability, it is almost trivial to construct different exploits for each

vulnerability. Furthermore, the nature of exploitation can be changed. Whereas one exploit might simply exit the vulnerable server cleanly, another may install a reverse shell, whereas another may launch a new process or thread.

The system can then be tested using each of these exploits, and the detections tallied. Not only are raw scores available, but the types of exploit/vulnerability combinations caught and missed can also be quantified. This gives the tester significant insight into the efficacy of the products under test (that is, is the exploitation being detected, or the post-exploitation behaviour and exploit payload).

Furthermore, if the exploitation of a vulnerability is missed, the tester can increase the size and 'noisiness' of the payload. Thus, products that have a post-infection detection filters can have their level of protection quantified.

FURTHER WORK AND CONCLUSIONS

The creation of new vulnerabilities in a program to which one has source-level access is relatively easy. Vulnerable code is easy to recognize, and when one can hand craft the source code and compiler options, writing a functional exploit is easy.

The primary weakness we see with our approach is that a program that uses crowd sourcing to determine 'normal' behaviour may have issues with our patched program, as it is different to other versions of the 'same' application. However, this is at least partially a real-world scenario; companies often customize software for their own purposes. While it would be ideal if our exploitable binaries were identical to those commonly found, we see no way around this limitation. Regardless, we do not believe this is a major limitation, as long as it is disclosed.

The next step for this approach is to actually build tests that use it. We hope to find a testing laboratory to collaborate with in the Fall. Our expectation is that results will be fascinating. In particular, it will be illustrative to compare results from our experiments with those of tests conducted using known exploits from an exploit framework such as Metasploit. Products that do well when using known exploits but poorly in our tests are likely to be detecting known signatures of specific exploits. This is not insignificant protection for the end-user, as hackers do make use of tools such as Metasploit, but it is not nearly as useful as one might think. For the well-run environment, patches should be pushed out as quickly as possible, and we are measuring the difference between this gap in protection (post exploit disclosure, while waiting to deploy a patch) and protection from a 0-day vulnerability itself.

We see this testing approach as a pragmatic and effective way of testing exploit detection capabilities. By carefully selecting the types of vulnerabilities and the modes of exploitation, a tester can conduct a repeatable set of tests without concern over impact of disclosure, vendor bias, or even cost of testing. The level of expertise required to create an exploit when you have hand-crafted the vulnerability yourself is not high, making it possible to test a range of defensive techniques.

As a closing thought, we make careful note that product testing shapes product design. A poorly designed test can and will entice vendors to cut corners with respect to 'real-world' security. For example, product design often requires trade-offs. A vendor may, for example, reduce the numbers of checks during certain operations in order to do better in a speed test, even if such a condition reduces security and rarely occurs in real-world use cases. Thus, good, solid testing needs to recognize the evolutionary pressure that testing places on product design decisions. The more closely a test conforms to a real-world scenario, the less likely it is that the test will persuade product developers to account for artificial use cases at the expense of real-world security.

REFERENCES

[1] Stevens, S. S. On the theory of scales of measurement, 1946.

[2] MRG Effitas, Real World Enterprise Security Exploit Prevention Test. Published Feb 2014. Available online through http://www.mrg-effitas.com/.

[3] Somayaji, A.; Li, Y.; Inoue, H.; Fernandez, J. M.; Ford, R. Evaluating Security Products with Clinical Trials. The 2nd Workshop on Cyber Security Experimentation and Test (CSET '09), 2009.

[4] Levesque, F. L.; Nsiempba, J.; Fernandez, J. M.; Chiasson, S.; Somayaji, A. A clinical study of risk factors related to malware infections. Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security (CCS '13). ACM, New York, NY, USA, 97–108, 2013.

[5] Fortify Software, Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors. http://www.hpenterprisesecurity.com/vulncat/en/docs/Fortify_TaxonomyofSoftwareSecurityErrors.pdf.

[6] Common Weakness Enumeration. http://cwe.mitre.org/about/sources.html.