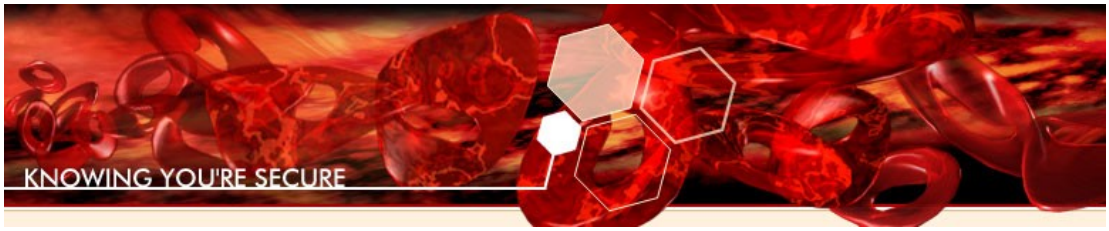

CANVAS Exploit Categories Explained

November 9th, 2009

By: Alex McGeorge, Immunity Inc.

IMMUNITY

Table of Contents

1 Introduction

One of the most common support requests we receive at Immunity is to explain the difference between various types of exploits that are found within CANVAS and the associated terminology. This document attempts to give the reader a very high level overview of the differences between exploit categories and when and where to use them. Most of the vocabulary used in this document is not CANVAS specific and is applicable for any offensive security product or project which deals in active exploitation.

1.1 Who is this paper for?

This paper is for anyone who new to the idea of exploiting software. Though even non technical readers will gain a better understanding of software exploitation after reading this paper, some familiarity with certain technical concepts will aid in their reading. Readers should be familiar with basic operating system, networking and programming concepts.

2 Vulnerabilities vs. Exploits

Vulnerabilities and exploits go hand in hand, exploits could not exist without vulnerabilities but not every vulnerability will have an exploit. A *vulnerability* is simply understood as a flaw in a piece of software either at the code or configuration level which allows a malicious user to affect some non-intended action. For our purposes software written to take advantage of these flaws are called *exploits*. Within the larger field of computer security this is not a hard and fast definition of what an exploit is but for CANVAS this understanding will suffice. Let's take a look at a practical example:

A software assurance tester has begun testing on a new network device. One of the many tests they will run is sending malformed ICMP Echo packets at the device. After extensive testing the tester finds a specific combination which will make the device reboot. The tester has just discovered a *vulnerability* in the device. Once some analysis of the packet captures has been completed, the tester discovers the specific sequence of packets required to force the device to reboot. They then use a scripting language to dispatch those packets on demand and now whenever the script is run the tester can reliably reboot the device. The software the tester has written is an *exploit*.

3 How do Exploits Work?

Exploits typically follow a simple high level pattern:

1. Manipulate the target program into a state where an attacker can tell the target

program what to do next

2. Provide those instructions on the next action to take
3. If possible, manipulate the program to continue normal operations

From this high level perspective any exploit that involves traversing the network will be more complex because we have to provide instructions to the exploited program to return results of the exploitation back to the attacker over the network. Local exploits don't have this same issue because their results can be observed directly by the local attacker. Simply put, exploits which utilize the network generally have more variables to account for when running.

4 Exploitable vs. Non-Exploitable

To understand exploits one must fundamentally understand how software is programmed. Vulnerabilities exist within functions, functions require certain conditions to be met before they are called. Therefore before vulnerable code is executed on the target certain conditions must be met. With sufficiently complex software sometimes functions are included but never called, sometimes an obscene amount of conditions must be met prior to the function being called in the specific way which triggers the vulnerable code and with memory corruption bugs sometimes the vulnerable code can be triggered but memory is in a very non-predictable state. These questions and a raft of others must be answered to decide if a vulnerability is exploitable or not, but these questions are common. The problem essentially boils down to, can an attacker control enough of the target environment to reach the vulnerable code and influence the target environment to achieve exploitation. Let's consider a followup example to the one in section 2.

Sally the software tester has reverse engineered the operating system of the target network device by examining a firmware update image for the device. She has found a vulnerability which may lead to a buffer overflow. However the device must be set to receive traffic over IPv6 which is a non-default option and in her testing environment none of the network devices which provide traffic to her target forward IPv6 traffic. In this scenario the vulnerability still exists because the flawed code is loaded on her device, however it is not exploitable in her environment because she can not meet the requirements to run the vulnerable code.

4.1 *Exploitable, but to what end?*

Vulnerabilities may be exploitable but the relative value of the exploit is influenced by what non-intended action can be accomplished with its use. Though again not a hard and fast rule for our purposes exploits typically achieve three types of non-intended actions: *Information Disclosure*, *Denial of Service (DoS)*, and *Code Execution*. Exploits which achieve *Information Disclosure* will be able to determine

some piece of information the software designers didn't intend to be available to users. Exploits which achieve a *Denial of Service* influence the target to such a degree that one or all of its services are no longer available to other users. Exploits which achieve *Code Execution* allow for an attacker to run code on the target. Again lets take a look at our fictional example from sections 2 and 4 to get a better idea of what these concepts mean:

Sally the software tester has fuzzed and reverse engineered her way to an impressive set of vulnerabilities for her target network device, after a bit of development work her exploits are finished. By sending the device a malformed HTTP Get request the device returns her a base64 encoded string that has been XORed with a set value, after decoding the string it reveals the device's exact firmware version as well as the function that was unable to process her request, this is a straight forward I*nformation Disclosure vulnerability*. From previous work, she was able to write an exploit to reliably cause a *Denial of Service* against her target device by sending an ICMP Echo request with a larger than expected packet size. Finally after more research against the device's implementation of SNMP she finds a reliable buffer overflow which allows her to load assembly into the target processes' memory which instructs it to connect to a waiting listener on her machine. This exploit leverages *Code Execution*.

5 Memory Corruption Exploits

Exploits come in two essential classes: memory corruption exploits and non memory corruption exploits. Programs rely on the memory accessible by their processes to be laid out in an expected way, typically by the program itself. To a certain degree, they also trust the data stored in that memory. If an attacker can influence the content of the memory used by the program, either by writing directly to memory or manipulating the program's functionality to set that memory to specific values for them, then they can cause the program to react in non intended ways when it uses that memory. This necessarily requires an attacker to use the program in a way that was not intended by the original developer, as such there is always a risk that the program will crash because the attacker is operating outside the scope of the intended (and therefore untested) functionality of the program.

Real World Example: MS08-067 (CVE-2008-4250) is an example of a memory corruption exploit.

5.1 *Exploit Reliability*

Typically when speaking on reliability we are necessarily talking about a memory corruption exploit. In order for exploitation to occur the exploit has to modify memory to suit its goals, sometimes the only way to modify memory in the target

program is to modify a lot of memory all at once. When forced into this technique typically pieces of memory required to keep the program from crashing are also overwritten. Exploits can work around this by repairing the damage they've done while taking advantage of the vulnerability, however because program memory is dynamic by nature what is overwritten is not always predictable or standard between one instance of the program to the next. Various techniques exist for increasing the predictability of memory layout or providing multiple chances for corrupted memory to be accessed, but they are beyond the scope of this paper. When an exploit is not reliable a failed exploitation attempt generally produces two outcomes: the program crashes or the program does not crash but our exploit doesn't achieve its goal.

Real World Example: MS08-001 (CVE-2007-0069) is an excellent example of inconsistent reliability. Immunity's exploit for this vulnerability is able to achieve a shell on the target host about 1/20 tries due to our code being executed in the kernel and the requirement to replace some of the data overwritten with the original values throughout exploitation.

6 Non Memory Corruption Exploits

This type of exploit is usually (but not always) seen in web applications . These exploits typically leverage vulnerabilities in configurations or situations where an attacker can supply code to be run by the application without manipulating memory.

A simple example of a exploiting a configuration vulnerability would be replacing a file executed on a schedule by another account with a file that grants you some new level of privilege. For instance, a Linux system administrator has a cron job set to execute a script in /tmp/sysadmin/logroller.py every day at midnight, this file is set to be world writable. Therefore an attacker only has to construct a replacement logroller.py to grant themselves account higher privileges, then wait for the system to execute it.

An example of supplying code to an application to be run without corrupting memory to do it, would be PHP's eval() function. This function allows data stored in a variable or other location to be run by the PHP instance. If user controlled data is passed to this function it would allow the user to execute arbitrary PHP.

Real World Examples: CVE-2005-2672 is an exploit similar to what was described as a configuration exploit. CVE-2007-4187 is an excellent example of of code execution without memory corruption, see the source code for joomla_eval.py within CANVAS for more information on this vulnerability.

7 Payloads, Callbacks and Listeners

In section three we covered at a very high level how exploits work, there's one last bit of terminology that needs to be addressed. A *payload* is simply understood as the

instructions provided by the attacker that the target will execute. Strictly speaking a payload can be any instruction however a *callback* is most common. A *callback* is a type of payload (the one used in CANVAS) where the host connects to the attacker to receive interactive instructions. A *listener* is a process which listens on a predefined network port for a *callback* from a target.

8 Exploit Categories

Finally, we're ready to discuss the different exploit categories! Software exploitation is a very complex topic, dozens of books have been written on it and we're only scratching the very surface of the topic here. This document gives you some of the basic vocabulary required to understand exploitation.

8.1 Remote Exploits

Remote exploits require no user interaction on the target host to exploit the vulnerability. These vulnerabilities exist in services and daemons on the target computer, the one basic requirement for all remote exploits is that the attacker must be able to reach the service on the target over the network. For example if we find a Windows Server 2000 machine that has been living in a network closet for 8 years and has never been patched but is not hooked up to any network, that machine is still vulnerable to a host of remote exploits but it is not exploitable by remote users.

Real World Example: MS09-050 (CVE-2009-3103) is an excellent example of a *Remote Memory Corruption Exploit*. The server service on Microsoft Windows hosts is involved in file sharing over the network and is always listening for new connections. Attackers who are able to reach it can leverage an vulnerability where user supplied data is used to determine the address of a function call.

8.1.1 Pre-Authentication vs. Post-Authentication

As we mentioned earlier, vulnerabilities exist in functions which must have certain conditions satisfied before they are called, in this case we're speaking of code paths. Consider an FTP service as an example, during login the service will not attempt to execute the function to parse DIR commands as the user isn't even logged in to the service! All services have some level of functionality available prior to authentication if only to negotiate which protocol will be used to authenticate. After authentication the attacker's attack surface increases, because they are now able to access functionality that was not provided to them prior to authenticating. Some remote exploits exploit vulnerabilities in functions that are only available to the user after they have authenticated, hence Post-Authentication.

Real World Examples: CVE-2007-1567 is an example of a pre-authentication bug as it is a stack overflow in the USER command which is utilized in the authentication

process. CVE-2000-0573 is an example of a post authentication bug, the SITE command in WuFTPD is not accessible without authenticating (even anonymously.

8.2 Clientsides

Clientsides require some user interaction in order to reach vulnerable code. Typically this means a user will open a file sent by an attacker or visit a website controlled by the attacker (essentially the same thing, just different programs to parse the files). The attack surface available to the attacker has now increased, all the code provided by the programs used to parse the data is now potentially available to the attacker. Including any programs those initial programs doing the parsing can call out to! Think about your web browser spawning an instance of Adobe Acrobat to display a PDF within the browser window automatically. In order to reach this state the user must help the attacker a little by loading the attacker's data. Because the programs used to parse the files are typically run with the privileges of the user, code that is executed is run with those same privileges. This is the reason for the security mantra of creating accounts with least privilege.

Real World Example: CVE-2009-2994 is a good example of a clientside, a heap overflow in Adobe Acrobat allows attackers to execute code with the privileges of the user reading the malicious PDF. Here the user must read the PDF before exploitation is possible.

8.3 Locals

Local exploits require that an attacker already have some level of local access to the target, typically as a simple user. Now, the attacker has access to essentially all the code they will ever have access to on the target. Obviously users with higher privileges may have access to even more code, but if you already have those privileges then you've already won. Locals are used when you need to elevate your privilege beyond that of a simple user. A question we sometimes get in support requests is “aren't clientsides executable locally?” Yes this is true, if the attacker is a user on the system they can load data with vulnerable applications as discussed in section 6.2. However typically these applications allow for code execution as the user running the program, since you already are that user, you have this ability innately.

Real World Example: CVE-2009-1185 is a vulnerability that is only locally exploitable on Linux platforms which allows regular users to assume the privileges of the root user.

8.4 Web Exploits

Web exploits are a category of remote exploits, they are meant to be used against targets you have access to remotely and launched against a service (HTTP) that is

always listening, therefore they are in the remote exploit category. In CANVAS Immunity distinguishes between web exploits and remotes for purely esthetic reasons. At the time of writing Immunity has close to 175 web exploits. They needed their own category within the GUI so that the remote category didn't get impossibly cluttered.

9 Remote Access: Memory Resident vs. Disk Resident

For exploits which are run over the network a unique concern arises when exploitation has been completed, maintaining remote access to the host. The solutions to this problem come in two types, memory resident code and disk resident code.

9.1 Memory Resident

Memory resident code is, as its name implies, executable code which is initially loaded into the memory space of the exploited process that allow for continued access to the exploited target. Essentially this establishes a network pipe directly into the exploited host's memory so instructions can be loaded dynamically.

9.1.1 Benefits of Being Memory Resident

The primary benefit of maintaining a foothold on a host in memory is the evasion of forensics. Most host based defenses rely on examining files written to disk to determine when a host has been compromised. By never touching disk an attacker can avoid detection using these methods. Malicious attackers have a high degree of stealth built into their techniques as penetration tests are meant to mimic malicious attackers, this provides the best simulation of an attack.

9.1.2 Drawbacks of Being Memory Resident

Resigning an attack to being only memory resident means that there's no persistence. If the exploited target reboots, the process our instructions live in crash, a network device between us and the target crashes or dumps its state table, or a host of other issues means that an attacker would have to re-exploit the host.

9.2 Disk Resident

In this scenario the code to maintain access to the host over the network is written to disk, this can take a variety of forms, sometimes the file is an executable (like a CANVAS trojan or the HCN rootkit), on Unix like hosts it may just be a simple shell script that wraps netcat.

9.2.1 Benefits of being Disk Resident

The benefit of being disk resident is, as you would imagine, one of the drawbacks of not being disk resident, namely persistence. The code used to achieve a callback is written to disk and therefore an attacker has the luxury of finding a solution to reactivate that code without having to re-exploit the host. Since it is generally advisable to only exploit the target host once (for more attack theory see our Unethical Hacking course) this provides an attacker with a reasonable solution

9.2.2 Drawbacks of being Disk Resident

The primary drawback of being disk resident is that it enables disk based forensics which is what a high majority of host based defensive products rely on. There are techniques to increase the level of effort for forensic investigations but short of zeroing out the drive the possibility exists that some or all of an attackers movements can be recovered. Disk Resident remote access is the most common because it is the easiest to conceptualize and write custom tools for. However a canny attacker with a good knowledge of operating system fundamentals and assembly will be able to construct attacks which never touch the disk, putting the burden of forensic investigation on in-memory forensics as well as network based forensics.

10 Vulnerability Assessment vs. Penetration Test

In recent years, “vulnerability assessment” has come to refer to non-intrusive scanning for known, potentially exploitable security issues. A “penetration test” often refers to a further step in this process - exploitation of a subset of these potential vulnerabilities.

Although Immunity does not endorse these definitions, scanning and exploitation technologies are complimentary. If we adopt the above terminology, vulnerability assessment helps identify low hanging fruit quickly over a large network segment. Complimentary to that, having a product who can exploit those vulnerabilities - performing a penetration test - is also valuable.

Running vulnerability scanning software against a host or network segment is a useful first step in assessing a target's potential vulnerability, however it is insufficient when considering more realistic attack scenarios. In order to assess the real vulnerability of a target, one must adopt similar techniques to an attacker. Real world attackers seldom run vulnerability scanning software against potential targets, however they remain successful in penetrating a target. This indicates that the above terms, while useful in their own right, do not live up to their names when considering how an attacker would approach compromising an actual target.

We find that the general formula for assessing vulnerabilities involves decreased automation and increased manual inspection, with the aide of select tools, in order to resemble real world attacks as closely as possible. Recipients of services such as security assessments, penetration tests and vulnerability assessments should hold their vendors to this standard in order to maximize the validity of the tests being performed.

11 Conclusion

This document has covered some the basic vocabulary you'll need to understand how to use CANVAS. In truth a book could be written to help new users understand *everything* they'd need to know in order to use CANVAS to its full potential from networking fundamentals through social engineering. This document covers some of the exploit specific basic concepts users typically have trouble understanding. Immunity offers classes to cover material such as this in much greater depth.

12 Resources

The following books and links will further aid help explain exploits for the interested reader.

Modern Operating Systems, by: Andrew S. Tanenbaum

Hacking: The Art of Exploitation, by: Jon Erickson

The Art of Software Security Testing, by: Chris Wysopal, Lucas Nelson, Dino Dai Zovi and Elfriede Dustin

The Art of Software Security Assessment, by: Mark Dowd, John McDonald and Justin Schuh

Immunity, Inc:

www.immunityinc.com

sales@immunityinc.com

Phone: +1 212 534 0857

Fax: +1 917 591 1851

1247 Alton Road, Miami Beach, FL 33139