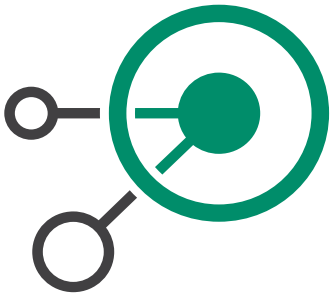

ERNW
providing security.

ERNW Newsletter 51 / September 2015

Playing With Fire: Attacking the FireEye MPS

Date:	9/10/2015
Classification:	Public
Author(s):	Felix Wilhelm

Enno Rey
10.9.2015 21:35:51

ERNW
providing security.

Table of Content

INTRODUCTION

In this paper, we present several now-patched vulnerabilities uncovered by a group of researchers in a FireEye NX device running the webMPS operating system in version 7.5.1.

The vulnerabilities presented here could allow an attacker to compromise virtual machine-based malware detection systems such as a FireEye device by triggering the analysis of a crafted exploit. Such an analysis can be triggered by sending an email to an arbitrary corporate address or by embedding the exploit code in a document (to-be) downloaded via HTTP.

All discussed vulnerabilities were responsibly disclosed and have been patched by FireEye. Please see the respective note released on Sep 08[1] for the official response from FireEye.

The remainder of the paper is organized as follows. In Section 2, we introduce the FireEye Malware Protection System (MPS) and feature set. Section 3 describes how a vulnerability (requiring prior authentication) in the management web interface can be used to gain access to the MPS operating system. In the following sections vulnerabilities in the Virtual Execution Engine (VXE) and the Malware Input Processor (MIP) are discussed in greater detail. Finally possible mitigation techniques are laid out.

It should be noted that changes to parts of this document were modified or removed after joint review with FireEye which might impact the readability/the train of thought to some portions of the document.

[1] https://www.fireeye.com/content/dam/fireeye-www/support/pdfs/fireeye-ernw-vulnerability.pdf.

1 Malware Protection System

The Malware Protection System (MPS) is a piece of software running on certain FireEye appliances. A core feature of MPS is the Multi-Vector Virtual Execution (MVX) engine, which is responsible for the dynamic analysis of samples and the detection & reporting of malicious behavior like exploits or callback communication.

One of MPS' main features is the detection of 0day exploits based on its dynamic analysis framework. The detection works by opening certain file types, including for example PDF, Office and Flash files, inside an instrumented virtual machine. If the files try to exploit a vulnerability by using typical techniques like heap spraying or ROP[2] chains or if a successful exploit triggers a process creation or a callback mechanism, this behavior will be detected and reported. In comparison to a traditional signature based approach this behavior analysis detects exploits much more reliably and is not restricted to known vulnerabilities.

The new capabilities gained by using virtual machine technology to detect malicious behavior also mean there are specific attack exposures that vendors must account for.

Usually administrative access to a FireEye appliance is possible using an HTTPS web interface and a restricted IOS-like CLI reachable over SSH. While these interfaces are sufficient for normal operation and administration they do not offer the possibility to get full access to the underlying operating system.

The next chapter describes how authenticated access to web interface can be escalated into OS access by using a vulnerability in the SSL certificate handling. Even though this vulnerability is not useful for a real world attack, it is a first step.

[2] *Return Oriented Programming*

2 Gaining Access

The FireEye web interface is a standard Ruby on Rails application and leaves a relatively robust first impression. An interesting functionality though is the certificate management that allows an administrator to upload new CA certificates.

2.1 Insecure CA Validation

Uploading random files results in an error message complaining about invalid certificate files and a failed certificate validation. Experience shows that such validation is often performed by traditional command line tools like *openssl* which then could make it a promising target for further digging.

After some trial and error, a way was identified to successfully trigger a command injection by uploading a CA certificate with content like this:

```
FOO"; echo 'test1234' > /tmp/test; echo "
-----BEGIN CERTIFICATE-----
MIIDtTCCAp2gAwIBAgIJAOtWde1RIp5yMA0GCSqGSIb3DQEBBQUAMEUxCzAJBgNV
BAYTAkRFMRMwEQYDVQQIEwpTb21lLVN0YXRlMSEwHwYDVQQKExhJbnRlcm5ldCBX
aWRnaXRzIFB0eSBMdGQwHhcNMTUwMzEyMTIxODI5WhcNMTYwMzExMTIxODI5WjBF
MQswCQYDVQQGEwJERTETMBEGA1UECBMKU29tZS1TdGF0ZTEhMB8GA1UEChMYSW50
ZXJuZXQgV2lkZ2l0cyBQdHkgTHRkMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIB
CgKCAQEAo0ofaG4JmPwlbeLMM5s39pHJwPvcoC/mWMv8T6YpKHUItMdUg8hFgsnL
Q+ypTVjVpmGGipj3gQnfVFvVebf4yhFEYjyqrj0i3vBIAcHpa7x0iDBtXmRf+6Os
j2UkzSikd3CYLrUNaQen4wx/HvFpb3Fll9AJqbcXUJ5mpPtbN+RC0zEARAJp6T1u
Ik9rWceChhYa/9mJiFG6Ktqq+9Yrt52hwh12H2tYQKc0T4QR4XRuH9D7iF/3JPyB
bG+kuWDU0MMEzCk7Z/o5XxufhUoRs1eL2C7COPWCiFkRzAZm5+YUBWfgOll0bCQL
hgiwR+PVC7omcDGCFsTp8UvArbX5+QIDAQABo4GnMIGkMB0GA1UdDgQWBBQmRWlD
q3/VUkVQSpUKaNo2KeYZjDB1BgNVHSMEbjBsgBQmRWlDq3/VUkVQSpUKaNo2KeYZ
jKFJpEcwRTELMAkGA1UEBhMCREUxEzARBgNVBAgTClNvbWUtU3RhdGUxITAfBgNV
BAoTGEludGVybmV0IFdpZGdpdHMgUHR5IEx0ZIIJAOtWde1RIp5yMAwGA1UdEwQF
MAMBAf8wDQYJKoZIhvcNAQEFBQADggEBABaAckAOXEPH7tY+ihRQxkYicWy6yNfS
cA3vVLpmNMlYqJLD+HYyoMM5c+rEQwnMZD5b/ywT3mV1sOiUdwnG7T8VnL4KdY9r
cH7DAub0N5w+I5oXsLEldzzpyuekvk9smQGB/ylwOXj6fwQcmVlaxWRmmzxq5sCQ
gCBQIRLvFzuVUYdWVt+W8vEqLi92V2ynQ009bzty6gZk2j8qvjmKYyXGkxfnyK/v
cwWhqDoWEbj7UvUJb0QRCZN2K37UqbTAZbXGgjIkFNCrqnJaX26H8GhNYa6qHnKB
x1xh+QtQnzOEIhtDyMkal+HKzuV3lQxssjDf1Gs8PCdZuRops9Biqp4=
-----END CERTIFICATE-----
```

Validation of this file triggered the creation of a file named /tmp/test; further access could then have been gained by various means, e.g. a simple reverse connect shell. Still it must be kept in mind here that the web interface is running in the context of a low privilege user account.

The whole CA file is stored in a temporary file and then passed to *openssl* for basic validation. If this succeeds, the file is split into single certificates based on the "—END CERTIFICATE—" line. It seems these chunks were then piped into

openssl again. Because the chunks are partially attacker controlled, a double quote breaks out of the *echo* argument and a command injection was possible.

The key issue here was the assumption that a valid certificate file only includes the certificates and nothing else. In reality, *openssl* happily ignores everything outside the BEGIN- and END CERTIFICATE blocks.

2.2 Privileged and Persistent Access

The shell gained by exploiting the described vulnerability was running under the low privileged *webui* user. The first step to get a reliable research environment is therefore the escalation to root and establishing stable remote access.

Interestingly the root user account is named admin, which was also the login name used to login into the web and SSH interface. In addition, a cursory look at the /opt/tms/bin/cli binary used as login shell for the admin users shows that it was the aforementioned restricted interface reachable by SSH.

After gaining root privileges, there were multiple ways to enable persistent access to the device. For example remounting the root file system as writable and replacing the *telnet* binary with a symlink to *bash* gave a stable way to jump from the restricted CLI to a full featured bash by executing the whitelisted *telnet* command.

3 COMPROMISING VXE

The Virtual Execution Engine (VXE) is one of the main components of FireEye's dynamic analysis. It's built upon a virtualized environment with several interfaces and user space and kernel drivers. One of those is of particular interest here.

3.1 Memory Corruption Vulnerabilities in libnetctrl_switch.so

The vxe binary directly passes raw network packets to the libnetctrl_switch library. In the first step, packets are parsed and stored in a PCAP. UDP packets that are not using the usual DNS destination port are forwarded to Snort for further analysis. All other packets are passed to one of two main analysis functions. One of the handlers is responsible for DNS traffic, the other one for everything else. The DNS handler is quite simple and just returns a fake DNS reply, while logging the requested hostname. The second handler is quite complex though.

For each newly contacted host, a new *addr* structure is allocated. Such a structure contains pointers to its previous and next neighbors, as well as information about the host IP address and DNS entry. In addition it holds an array of 10 *port* structures, to store state information for each contacted TCP port.

When a packet is sent to a new TCP port, the next free slot in the port array is chosen. If no free port is available because all 10 TCP destination ports are already used, the analysis is aborted. Otherwise, a state machine is initialized by analyzing the TCP payload: In the first step the first twelve bytes of the TCP data are converted to uppercase letters. They are then checked against a number of built-in strings that occur in supported protocols. For example, "GET" indicates an HTTP request, while "NICK" is used by IRC and "PASV" by FTP. Discovering such a string will initialize the first state of the state machine that is used to reply in a semi realistic way.

Using the sprintf function on a fixed size stack buffer without any length restrictions is, well, dangerous. Simply sending a string longer than 500 characters as a nick name will trigger a buffer overflow and crash this code.

While modern security features like stack cookies could render this bug useless for an attacker, we were unable to identify those checks in the version analyzed. Partial control over RIP can therefore be achieved. However, developing a full exploit for this issue is tricky, due to the facts that nick_name can only contain alphanumeric characters and no controlled partial overwrites are possible because the last bytes of the buffer are outside the control of an attacker.

One interesting target is a function called get_value. It copies a string from data to value, as long as it does not contain newlines, spaces or a null byte. Again no length checking is performed, making any caller which does not ensure that the value buffer is sufficiently large vulnerable to a buffer overflow vulnerability.

In the case of the IP handler functions discussed before, *get_value* is called when some of the initial trigger strings were detected. After matching on the string "JOIN" for example, *get_value* is called with value pointing to the *join_info* field of the aforementioned port struct and data pointing to the TCP packet data following the match. It turned out that join_info is only 1024 bytes long, allowing an attacker to overflow the buffer by sending "JOIN" followed by a long string. The same issue exists for the other *_info fields in the port structure.

In practice the length of the overflow is only restricted by the Ethernet MTU of 1500 bytes. Subtracting the lengths of IP and TCP header this still leaves enough potential to overflow more than 400 bytes.

join_info is the most interesting target because it is near the end of the port structure and overflowing it allows the corruption of the tcp_seg and state fields, as well as data stored behind the structure. Of course a corrupted TCP segment number does not offer any interesting advantages. The state field is used as an index into the state machine function table but is always checked against a small upper limit and therefore uninteresting.

Instead one can target data stored directly after the port structure in memory: If the join_info buffer is part of the last structure stored in addr->portTable the overwrite will corrupt the list_entry structure stored directly behind it. The list_entry structure contains a prev pointer and a next pointer and by manipulating them one can trigger an arbitrary memory overwrite as well as full code execution.

The basic outline for triggering an exploitable crash is the following:

- Trigger the analysis of a malicious executable.
- Connect to 9 different TCP ports on a randomly chosen host sending random data.
- Connect to another TCP port on the same host and send a "JOIN A*1048" followed by the new values for prev and next.
- Connect to a second randomly chosen host. This will trigger a memory write to the address specified by the next pointer and crash the target.

It was evidenced that this bug could be used to fully break out of the virtual environment and execute code on the host system. While a 100% stable exploit is not easy to achieve, the impact of such a breakout in real world exploitation is not trivial.

On the positive side, the nature of the bug introduces some dependencies on the MPS version of the target. In addition, the log entries generated by such an exploit are relatively noisy, decreasing the value of the bug. Hence in the next section a vulnerability in MIP will be shown that was 100% reliable against unpatched versions of MPS and did not leave any suspicious log entries on the system.

4 ATTACKING MIP

The Malware Input Processor (MIP) is responsible for orchestrating all static analysis actions performed on a sample.

Because malware or exploits can be compressed and embedded into archives, decompression is one of the first steps MIP performs. There is a high number of supported archive formats including well known file types. As already mentioned, all actions performed by MIP are executed directly on the MPS host system. This is true for the extraction process as well.

4.1 7zip Directory Traversal

The extract_ar.py script responsible for archive extraction performs the following call to decompress an archive:

```
subprocess.call(['/usr/bin/7z', 'x', '-y', dest_arg, pass_arg, archive_name])
```

As already publicly documented[3], the 7z extraction utility follows symbolic links included in the archive. This means that a malicious archive that is extracted in a harmless sub directory of the mip process, can perform a directory traversal during extraction and store files all over the file system. Due to the fact that the "-y" parameter is also passed to 7z, already existing files can also be overwritten. The only limiting factor for this vulnerability is the limited file system access (privilege-wise) of the mip user.

4.2 File Write To Code Execution

When looking at all writable directories, one particular entry is very interesting:

```
[admin@fireeye ~]# ls -la
total 308
drwxrwxr-x  5 admin contents  4096 Jun  8 16:57 .
drwxr-xr-x 40 admin root      4096 Jun 25 16:42 ..
-rw-r--r--  1 admin root     90010 Apr  2 13:24 OleFileIO_PL.py
-rw-r--r--  1 admin root      3498 Apr  2 13:24 dmg.py
-rw-r--r--  1 admin root      3876 Apr  2 13:24 doc.py
-rw-r--r--  1 admin root      4480 Apr  2 13:24 docx.py
[snip]
-rw-r--r--  1 admin root      3349 Apr  2 13:24 xls.py
-rw-r--r--  1 admin root      4326 Apr  2 13:24 xlsx.py
```

Here some directory was writable by all users in the contents group, which also includes the mip user. Even though all files in the directory were only writable by root we could simply overwrite one of them using the symbolic link approach, because the directory itself was writable. Apparently some python scripts were used to perform file specific analysis operations. For example once a new DOC file is analyzed, a script called doc.py is executed. A RTF file would trigger the execution of a similar script. By overwriting one of these scripts with our own version, one could therefore execute code with the privileges of the mip process when the next sample with the right file type arrives.

[3] *https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=774660*

In conclusion, this vulnerability allowed reliable code execution on unpatched MPS versions which could be triggered by a specific file. The last thing that was left to do is to escalate privileges from the mip user to root/admin.

4.3 Local Privilege Escalation

cms_agent.rb is a ruby script running with root privileges and listening on the local TCP port 9900. While its name suggests that it is used for centralized management, it was an interesting target for local privilege attacks.

The script creates a dRuby server to listen on port 9900. dRuby stands for distributed ruby and is a RPC mechanism allowing the execution of almost arbitrary methods in the remote process. In the case of cms_agent.rb most functionality is restricted and a remote client can only call a few methods. One of them was the method mdreq_exec that executed the mdreq binary with a supplied string as first parameter:

Because no validation was undertaken, a command injection could be performed, offering a path to execute commands with root privileges. So by combining several vulnerabilities, an attacker could potentially gain complete root privileges on an unpatched appliance. Because no memory corruption vulnerabilities were involved this bug chain was 100% stable and exploitable.

5 CONCLUSIONS

In this paper the potential attack surface of virtual machine-based malware detection systems, in this case FireEye's Malware Protection System, is examined. In the scenarios assessed some vulnerabilities were identified that could have lead to a full compromise of an unpatched appliance.

From a researcher's perspective the following measures come to mind to mitigate the impact of the types of issues mentioned:

- Compiler hardening.
- Running the static analysis process in a virtualized setting.
- Hardening of local privileged processes.
- Implementation of parsing code in memory safe languages.

Even with these measures in place, a certain chain of (bug) events could lead to a vulnerable state of in virtual machine-based malware detection systems.

All vulnerabilities laid out above have been patched by FireEye, please see the vulnerability summary released on Sep 08 2015[4].

[4] https://www.fireeye.com/content/dam/fireeye-www/support/pdfs/fireeye-ernw-vulnerability.pdf.