The Master and Parasite Attack

Lukas Baumann, Elias Heftrig, Haya Shulman and Michael Waidner
Fraunhofer Institute for Secure Information Technology
Darmstadt, Germany

***Abstract*—We explore a new type of malicious script attacks: the *persistent parasite attack*. Persistent parasites are stealthy scripts, which persist for a long time in the browser's cache. We show to infect the caches of victims with parasite scripts via TCP injection.**

Once the cache is infected, we implement methodologies for propagation of the parasites to other popular domains on the victim client as well as to other caches on the network. We show how to design the parasites so that they stay long time in the victim's cache not restricted to the duration of the user's visit to the web site. We develop covert channels for communication between the attacker and the parasites, which allows the attacker to control which scripts are executed and when, and to exfiltrate private information to the attacker, such as cookies and passwords. We then demonstrate how to leverage the parasites to perform sophisticated attacks, and evaluate the attacks against a range of applications and security mechanisms on popular browsers. Finally we provide recommendations for countermeasures.

I. Introduction

Malware is the most significant threat to computer users; and, from all types of malware, the greatest threat are *advanced persistent threats (APTs)*, which may reside in the victim's computer for long time without detection. However, installing malware - and APTs - on a victim's computer can be challenging. Either the attacker has to trick the user into installing a malicious software, or a vulnerability in user's system or applications must be exploited to install the malicious software without the user's permission.

In contrast, browsers automatically download and execute code from remote websites. Unlike with malicious software, such code runs in a restricted *sandbox*, rather than the 'native' code of malware. The goal of the sandbox is to prevent attacks by automatically downloaded code. Furthermore, remotely-downloaded Javascript code executes only while the browser visits the relevant website, i.e., *ephemerally* rather than *persistently*, and stays in the browser for as long as there is place in the cache and the website does not serve a new version of the same object. Hence, any malicious computation, such as abusing users' compute power for crypto-currency mining [24], can affect the users only as long as they visit the website or until the script is evicted from the cache.

In summary, Javascript and other remote code executed in a sandbox is believed to be a relatively minor – in contrast to user or root level malware – ephemeral (non-persistent) threat. However, in this work, we show that remotely-downloaded Javascript can be *persistent* and poses a serious threat. In particular, we show that attackers can perform an attack so that these scripts are executed permanently in victims' browsers and then actively spread this attack to other domains. The first step of our attack includes injecting and spreading a malicious script, which we call *parasite*. We then show how these scripts can be abused by the attacker to perform various attacks against applications, such as stealing clients' credentials on popular web sites or bypassing a two factor authentication. We evaluate the effectiveness of these attacks on popular browsers. Typically such attacks should not be possible: a script coming from an attacker would be restricted by the Same Origin Policy (SOP) and hence should not be able to access data of objects coming from other domains. *To bypass the SOP the attacker camouflages the malicious (parasite) script to appear as if it originated from a real origin website.* We develop methodologies to keep the parasite in the victim's browser over long time periods and to control it to launch attacks against different applications, even after the client moves to a different (e.g., home) network. The parasite uses modules we developed for *Command and Control (C&C)* to the attacker and for *propagation*. We demonstrate attacks that we implemented using the parasites and provide results of our experimental evaluation.

Attack Overview. The attack consists of four modules: (1) cache eviction, (2) injection into transport layer, (3) parasite construction and finally (4) applying the parasites to launch sophisticated attacks.

As a first step, the attacker performs cache eviction (Section IV), to remove cached objects of popular target domains, then causes the browser to issue an HTTP request to the target website. The attacker injects a spoofed TCP segment with the infected object into the HTTP response from the server, which delivers the malicious payload into victim's cache (Section V). The injection can be performed with a remote attacker by launching DNS cache poisoning [33, 16, 34, 17, 13, 5, 15, 19, 21, 1, 44, 5]. As soon as the parasite is cached in the victim's browser, it starts infecting other domains in the cache and establishes a C&C to the remote attacker to carry out advanced attacks against applications (Section VII).

Contributions. Our goal is to explore the attacks against applications that remotely controlled parasite scripts can launch and the feasibility of constructing such parasite scripts in the wild. Our contributions can be summarised as follows:

- We identify websites which use persistent objects. We measure the persistency and prevalence of these objects on popular websites. We found that more than 87% of the websites have at least one object persistent over a period of 5 days. We modify these objects attaching to them malicious

parasites.

• We develop methodologies to force the browsers to keep our parasite scripts in the cache even after the victim stops visiting the website whose object was infected with a parasite, and even after the victim moves to a different network.

• We devise techniques to bypass SOP, allowing the parasites to propagate between different domains and caches. Our techniques can be systematically launched via TCP injection in contrast to previous work which exploited bugs to bypass SOP [27]. Since our approach leverages the standard behaviour of the systems it is much more difficult to block.

• We develop and evaluate command and control channels for communication between the attacker and the parasites.

• We developed a taxonomy of popular caches and evaluate our attacks experimentally against them. These are caches that can be infected with our parasite and then used to attack the applications.

• We developed a taxonomy of the exploits against popular applications which we attacked with our parasites botnet. The applications range from financial and banking systems to hardware components on devices (such as camera and mic).

Organisation. In Related Work, Section II we review research relevant to our work. In Section III we explain our attacker model. In Sections IV, V and VI we present the implementation of our attacker and report results of our evaluations in the Internet. In Section VII we demonstrate how parasites can be leveraged to attack a wide range of systems and applications and provide recommendations for countermeasures in Section VIII. We conclude this work in Section IX.

II. Related Work

Malicious JavaScript attacks. Due to policies like SOP [30], CORS [40] and CSP [36] execution of scripts is sandboxed to the specific domain. Nevertheless, there have been attacks that demonstrated that scripts can utilise side channels to read memory or even change content of memory [23, 14, 35]. Furthermore, malicious scripts do not necessarily need to break out to cause harm, as they have access to the CPU and may utilise this computing power for cryptojacking [12]. Prior Javascript attacks exploited bugs in browser's implementations of SOP [8, 3, 18, 29]. We demonstrate attacks which do not exploit a bug in the SOP, but combine transport layer attacks in tandem with browser's cache infection to bypass SOP restrictions. This makes our attacks much more difficult to block since they exploit the correct behaviour of the protocols.

There has been work on how to circumvent SOP and CSP by loading images, that contain JavaScript, e.g., [20]. An attacker would still need access to the domain as his image has to be loaded there, to unleash the full potential of JavaScript. Same applies to the SOP. In contrast, our attack infects the domain directly and supplies new methods to create communication channels outside the SOP. The CSP is able to reduce the impact, but as the scripts are executed within the domain, a wide range of attacks may be executed.

Browser cache poisoning. There has been research on how to poison the cache of a browser in different ways, e.g., [7, 39, 9]. These show, that caching is a problem at the clientside as the server is not contacted once an element has been cached. This is extended by even injecting malicious code into objects loaded via HTTPS due to users ignoring the certificate errors or supplying malicious browser extensions for the clientside. In this work we show how to create a persistent network of malicious files without a user browsing these while being exposed to an attacker.

Finally, we perform the first experimental study of the attacks that parasite scripts can launch against different applications. We also evaluate the infection, the propagation and the communication of the parasites on popular browsers.

III. Attacker Model

Our attacker consists of a master which injects malicious scripts - we call *parasites* - into the browser caches of the victim clients. The master then controls the parasites and uses them to launch sophisticated attacks against the victim clients.

Master can eavesdrop on the packets exchanged by the victim client but cannot block or modify them. Such an attacker can be another client on a public Wireless network. The master sees the TCP source port and the TCP sequence number in the segments sent by the client and hence can craft correct response segments impersonating the server, without the need to guess these parameters. Our attack proceeds by injecting TCP segments into the TCP connection between the victim client and the target website.

Parasites are scripts coming from the legitimate website modified by the attacker to include a malicious behaviour. The parasite is injected by the master into the communication between the client and the website. It is then cached by the victim client's browser. We show how to construct the parasite so that it stays in the browser persistently over long time periods, even when the victim client shuts down its device or changes network. The parasite propagates and infects multiple other domains in the victim's browser's cache. The parasites communicate between themselves and the master and execute commands and attacks on behalf of the master.

IV. Eviction from Browser's Cache

To infect an object from some website with a malicious script of the attacker, the attacker has to cause the client to issue an HTTP request for that object. Typically objects from popular websites will be cached in victims' browsers, hence during repeated visits to a website the client will not issue requests for objects that are already cached. As a result the attacker cannot inject its malicious script.

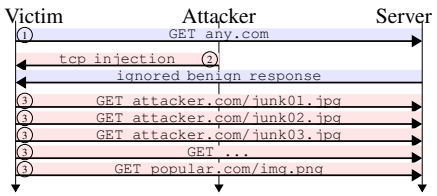

Fig. 1. Cache eviction. Legitimate messages are blue, attack are red.

To force the browsers to retrieve the newest copy of the object we perform an enforced cache cleaning. We developed

a *cache eviction* module, which removes (evicts) all the objects from the browser's cache. Cache cleaning is illustrated in Figure 1 and proceeds as follows: When a user establishes a connection to some website, the attacker injects a spoofed TCP segment (a small inline script), impersonating an authentic TCP segment from the website, into a TCP connection between the user and the website. The injection is illustrated in Figure 1 ②. This script dynamically loads requests for junk objects (images) in the attacker's domain. The responses populate the cache with the objects and supplant (older) cached elements. The *cache eviction* procedure ensures that all further requests for objects generate new retrieval requests from the web server (Figure 1 ③), which responses get modified by the attacker in the next step.

Evaluation. We evaluated cache eviction against popular browsers[1]. The browsers differ in cache types, sizes and cache eviction methodologies. We also investigated, whether they allocate memory per domain. If cache capacity is shared between all domains, even if a victim cached objects of `a.com`, they may be evicted upon visit to `b.com`, for example. The results are shown in Table IV. Eviction from Chromium-based browsers and Firefox is performed easily and efficiently. It has been observed, that Firefox, while evicting cache, may experience reduced responsiveness due to overloaded memory and disk cache. Internet Explorer behaves differently: it appears to allocate more and more space to the memory until the operating system shuts down processes due to low free memory.

Browser	Version	Ev.	I.D.	Size	Remarks
Chrome	81.0.4044.122	✓	✓	320MiB†	†from Chromium
Chrome*	81.0.4044.122	✓	✓		*incognito mode
Edge	84.0.522.59	✓	✓	320MiB†	
IE	11.1365.17134.0	×	×	330MB	DOS on memory
Firefox	75.0	✓	✓	256MB	performance impact
Opera	68.0.3618.56	✓	✓	320MiB†	

TABLE I

EVALUATION OF CACHE EVICTION ON POPULAR BROWSERS. 'EV.' IS EVICTION, 'I.D.' IS INTER-DOMAIN, AND 'SIZE' REPRESENTS DEFAULT CACHE SIZE.

V. INJECTION OF PARASITES INTO TCP CONNECTION

Injection of scripts can be done with an off-path attacker that is not located on the same network as the victim, e.g., via DNS cache poisoning or BGP prefix hijacking [5, 4], or via injection of TCP segments by inferring the ACK number and sequence number (SN) [9]. However, since the focus of our work is on evaluating the attacks that such scripts can launch against different applications when running on popular browsers, we perform the injection of script in an eavesdropping attacker mode into the browsers. We assume that our eavesdropping attacker is located on the same wireless network as the victim, e.g., on a public WiFi network.

OS	Chrome	Firefox	IE	Edge	Safari	Opera
Win10	✓	✓	✓	✓	✓	✓
MacOS	✓	✓	n/a	n/a	✓	✓
Linux	✓	✓	n/a	n/a	n/a	✓
Android	✓	✓	n/a	n/a	n/a	✓
iOS	✓	✓	n/a	n/a	✓	✓

TABLE II

TCP INJECTION EVALUATION. NO SUPPORT BY AN OS MARKED 'N/A'.

[1]https://www.w3schools.com/browsers/

Setup. The master monitors the communication on the network, waiting for an HTTP request to one of the objects he has prepared in advance, and injects a TCP segment containing the malicious response once such HTTP request is detected. To ensure that the TCP segment is accepted and correctly reassembled, the master must set the correct TCP destination port, TCP sequence number (SN) and offsets - these fields he can adjust from the HTTP request packets that the victim client sends. We implemented and evaluated injection of TCP segments in communication of popular browsers, and confirmed the attack to be effective, independent of the browser and OS; the results are listed in Table II. The messages exchange diagram for injecting a TCP segment is illustrated in Figure 2.

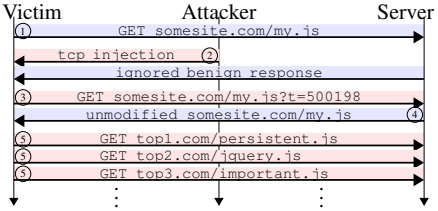

Fig. 2. Cache infection attack. The blue packets indicate responses from the genuine webserver, red packets are those injected by the attacker.

Attack. In step ① the victim sends a request for *my.js* file. The attacker sends a response containing a malicious script (a parasite) in step ②. The parasite contains the same name as the original script that was requested by the client. The functionality of the webpage may be modified since the authentic script was not loaded - this may be detected by the victim. To avoid detection, the parasite (i.e., the newly infected script) issues a request to the website to load the original script using a different URL (the with an ignored request parameter), steps ③ and ④. This request satisfies the Same Origin Policy (SOP) limitation and is allowed by the browsers. The parasite subsequently initiates propagation to other webpages and issues requests to popular webpages, in steps ⑤ and on, which in turn get infected by the attacker as in step ①. This results in multiple parasites, each corresponding to an infected script from one popular webpage. This guarantees availability of this puppet - namely, every time any of these popular websites are accessed, the parasite (corresponding to the requested website) is invoked.

Discussion. One of the main countermeasures against TCP injection attacks is to employ encryption. Although this fact has been known for a long period of time, our measurement study found that 21% of the 100,000-top Alexa websites do not use HTTPs and almost 7% of the websites use vulnerable SSL versions (SSL2.0 and SSL3.0). Furthermore, even websites supporting SSL/TLS for communication may be compromised. Recent works demonstrated that off-path attackers can trick Certificate Authorities (CAs) into issuing fraudulent certificates, [4, 5]. If our attacker uses a fraudulent certificate for some target domain it can similarly inject spoofed TCP segments into communication with that domain.

Often even simpler attacks suffice. We evaluated the 15K-top Alexa domains and found that from the 13 419 HTTP(S)

responders 67.92% did not provide HSTS headers at all, and only 545 were contained in Chrome's HSTS preload list, leaving up to 96.59% of the domains vulnerable to SSL stripping attacks.

VI. The Parasite Design and Implementation

In this section we show how to select scripts that should be infected with parasites, how to ensure they stay persistently in victim browser cache, how to develop methodologies to allow parasites to propagate between different devices and domains and how to develop a Command and Control channel to communicate between the parasites and the attacker.

A. Infecting Objects with Parasites

The attacker's goal is to select a script from some legitimate domain and to infect it with a parasite. The objects on websites can be changed, renamed or even removed. In that case, the control over the parasite-script instance is lost since it will never be invoked and will eventually be removed from the cache. The goal of the attacker is therefore to select such a script that will ensure long term control over the parasite in the browser of the victim client. We achieve this using two key observations: we select objects which do not change often and in those objects we set the headers so that the cache keeps the injected script for the longest possible time duration.

Selecting persistent scripts. Which script should we infect in order to guarantee persistency? Ideally the attacker would search for scripts that do not change often and whose names are stable over long time periods. To identify such scripts we develop a web crawler to collect statistics over 15K-top Alexa pages. For all objects on these pages, we collect hashes over the files and names, and store them. The web crawler ran daily over a period of 100 days. At the completion, we perform an analysis over the collected data, the results are plotted in Figure 3. For instance, in Figure 3 for a window of five days about 87.5% of the websites use at least one persistent object (excluding inline scripts), namely, object that is not renamed over a period of five days. After approximately 100 days, 75.3% websites are using at least one persistent JavaScript file, which has not been renamed.

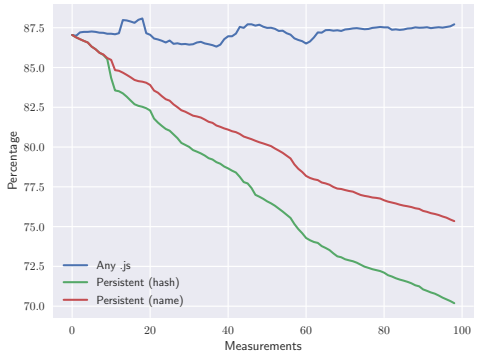

Fig. 3. Persistency measurement over 100 days.

These scripts are perfect targets to be infected with parasites for our persistent botnet, as they are accessed frequently due to the popularity of the websites from which they are served. Figure 3 also shows that JavaScript files might change in content, while not necessarily in name. If the name was changed within the website, it is not usable for our attack anymore as browsers' caches use names of files as keys and hence, we focus on the name based persistency factor. Using these statistics we are able to select files which remain persistent over time. We use these files as the potential targets for camouflaging our parasites.

Setting parasite caching headers. In the JavaScript of the parasite the caching related headers are set to ensure that the browser of the victim keeps the modified copy of the object as long as possible in the cache; the cache duration is set by HTTP headers like the `Cache-Control` header.

Infecting scripts with parasites. When identifying a request for a persistent object in one of the domains of interest the attacker injects a TCP segment containing the original objects attaching the parasite script to the end.

The attacker loads the original object that would normally be included in the target page. Then the parasite object is created by expanding the original file. "; PARASITE_CODE;" is appended to the end of the corresponding original JavaScript file. For HTML files, a "<script>PARASITE_CODE</script>" tag is inserted before the closing "</body>" tag. However, inserting it in the HTML file is optional so as not to violate any Content Security Policy. The variables and function identifiers in the parasite code have been chosen so that there is no conflict with the target applications.

Afterwards, every time a client makes a request to the server checking for "freshness" of the infected object, the request is manipulated to ensure that infected object is refreshed.

Requesting the infected objects. Every time the infected object is requested by the website our parasite script is invoked instead of the original script. The client's request is manipulated. Headers are set which signal to the server that the client has not cached any data. This prevents the server from responding with a 304 status code. The manipulated request is forwarded to the server. The server's response is also manipulated. If the response is an HTML or JavaScript file, the malware is injected. The original function is preserved by attaching it to the end. We also perform validity checks on the server. The cache headers are adapted, so that the data gets in the cache of the client and remains there for as long as possible. In addition, security headers are removed. This makes it possible to cross-infect other domains. The client caches the parasite and it is executed every time the corresponding resource is loaded. Furthermore all HTTP caches between attacker and victim will be poisoned with the manipulated JavaScript files.

B. Propagation

We developed two methods of propagation for the parasites: the parasites can propagate to other domains on the same victim device and the parasites can propagate to other devices.

1) Propagation between domains: After infecting an object from one domain, the parasite can propagate to infecting objects from other domains on the same victim browser. In

this section we provide two methods of propagation between domains that we evaluated.

Propagation on the same device via shared files. For instance Google Analytics which our measurement found to be used by 63% of the 1M-top Alexa domains [6]. Infecting this JavaScript file in the browser cache therefore leads to the parasite being executed on a large number of domains.

Propagation via iframes. For propagation via iframes, the parasite loads the target domains via iframes into the DOM. The browser then loads all the resources associated with these domains.These objects are infected with parasites. This is only possible because all corresponding security headers (more about this in Section VIII) are disabled. This cross-infection is demonstrated in the demo video[2]. In this video we show, how the visit of a well known and popular site on an insecure Network can lead to the infection of other sites like online banking and web mail, that are not even accessed and used during the active attack.

2) Propagation between devices: We experimentally measure how the parasites can propagate between caches of different popular devices. Propagation between devices is made possible by shared network caches but can also be done due to vulnerabilities in web services (e.g., via XSS attacks). The principle is to attack victims behind shared caches: when a victim receives an infected object from the server, all the caching proxies on the way cache that manipulated object. After that all other victims using these proxies receive the malicious cached object.

Browser	Ctrl+F5	clear cache	clear cookies
Chrome	×	×	✓
Firefox	×	×	✓
Edge	×	×	✓
Opera	×	×	✓
IE	n/a	n/a	n/a

TABLE III
EFFECTIVENESS OF REFRESH METHODS TO REMOVE OBJECTS STORED WITH CACHE API (NOT SUPPORTED BY IE).

We perform evaluation of parasites' injection into popular caches in the Internet and show that the attack has a substantially wider application scope than merely being applicable to end hosts' (clients) browser caches. The caches that we found vulnerable to our attack are listed in Table IV. The vulnerability in that case is that the browser cache is shared between multiple sites. As a defence some modern browsers offer separate caches for each 'calling context'. Network caches (e.g., on ISP or on local network) do not support such an isolation and can therefore be seen as a shared resource hence enable the associated attacks. Besides side channel attacks, it is also possible to use network caches as a way of infection. If the entry for a client in the cache is infected, it automatically affects all other clients connected to the cache. Since it is a design feature of network caches to minimise resource usage by sharing resources, but no security mechanisms are provided on the protocol side, all network-based HTTP(s) caches are vulnerable to our attacks. To prevent this, an isolation can be applied in the cache per client, which however would harm performance. Fixing the software of the network cache is not trivial. Injection attacks against reverse proxies (e.g., on CDNs) also affect all users of the infected proxy.

In our evaluations, we saw that different types of caches have different persistence strategies. The persistence of the browsers with regard to Cache API[3] was experimentally evaluated using a lab validation server. Our evaluations demonstrated that in all the browsers, cleaning up the cache does not suffice to prevent the attacks. In particular, the cookies must also be deleted in order to remove the parasites from the cache; see Table III.

C. Command & Control Channel

After the victim disconnects from the network on which the initial infection was made, the master uses a C&C channel to communicate to its parasite instances on the victim. We design and implement a bi-directional C&C channel for communication between the different parasites on a victim and for communication between the parasites and the remote attacker. Our channel enables sending commands to the bots and retrieving data from the victims. The process is illustrated in Figure 4.

Instead of relying on known protocols and features, which can be blocked, such as CORS, we design our own communication protocol.

We use HTTP information leakage combined with cross requests to allow a remote attacker to create a channel with the parasite scripts in different domains. Our channel is based on the dimensions of a cross image requests. When a cross image request is performed, most image properties are hidden, but the image dimensions are visible. This is needed to adapt the page to the image proportions. For communication we are transferring multiple images, encoding the information in the width and height of the images. Our experiments show that once the dimension is over 65,535, the browsers will downgrade it to this value. Therefore, we can transfer in each image 2 values between 0 and 65,535. Consequently, each image contains 4 bytes of encoded data. We use `SVG` format,

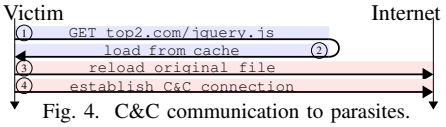

Fig. 4. C&C communication to parasites.

so that the total image size remains small, keeping down the overhead bytes for transferring these 4 bytes of data. An `SVG` image, having no actual content, is of size 100 bytes, and in our experiments, using a client which sends requests for multiple images simultaneously, we achieve a communication channel of 100KB/s, from the master's server to the parasite.

The communication channel in the other direction, from the parasite to the master's server is as follows: we use HTTP requests, where the URL, or even the URL get/post parameters are the encoded data, and hence with no bandwidth limitations.

[2]http://52.144.44.214/demo.html

[3]https://developers.google.com/web/fundamentals/instant-and-offline/web-storage/cache-api

Location	Type	Instance	HTTP	HTTPS	Comment
Caches on Victim Host - Client-internal Caches	Browser Cache	Desktop	✓	✓	
		Smartphones [26]	✓	✓	
Caches on Victim Network - Client-side Cache	Transparent Proxy	Squid	✓	✓	
	Web Filter	Cisco Web Security Appliances	✓	✓	AsyncOS 9.1.1
		McAfee Web Gateway	✓	✓	
		Citrix NetScaler [10]	✓	‡	
		Barracuda Web Filter	✓	×	
		Blue Coat ProxySG	✓	×	
	Firewall	Sophos UTM	✓*	✓	*community-documented
		Fortigate	✓	✓	
		Barracuda F-Series	✓	×	
		Cisco ASA	✓*	×	*via redirect
		pfSense	✓*	×	*via squid module
	Transport	Airplanes [31, 32]	✓	‡	
		(Cruise) Vessels [2, 41]	✓	‡	
Remote Caches - Backbone and Server-Side Caches	Reverse Proxies HTTP Accelerators	CDNs	✓	✓	
		Varnish HTTP Cache	✓	✓*	*when used with separate SSL Offloader
		F5 Big-IP WebAccelerator	✓	✓*	
		SiteCelerate	✓	✓*	
	Web Application Firewall	GoDaddy WAF	✓	‡	
	ISP	CacheMara	✓	×	
	Mobile Network	LTE Network[28]	‡	×	
		5G Networks [43]	‡	×	with MEC

TABLE IV

EVALUATION OF CACHES IN THE WILD. CACHING ENABLED BY DEFAULT (✓), OPTIONAL (✓), NOT SUPPORTED (×) OR SUPPORTED BY ARCHITECTURE MODEL BUT FUNCTION NOT PUBLICLY DOCUMENTED OR IMPLEMENTATION-DEPENDENT (‡).

VII. PARASITE ATTACKS AGAINST APPLICATIONS

The attacks we developed and evaluated are application dependent. We categorise the attacks per target: against browsers, against operating systems and against networks. The attacks, along with their type, the targets, the exploits and the requirements are listed in Table V. We incorporated the different attacks into the parasite via the following modules: (1) a module that reads the browser data such as current URL, user agent, cookies, the local storage; (2) a module for extracting protected browser data, for example, microphone capturing; (3) a module for spreading the parasite based on customised phishing, similar to Emotet [37]; (4) a module that extracts login data, e.g., from Google, Facebook or online banking applications. The parasites use the URL to detect on which website they are currently running, then execute the corresponding attack modules.

The vulnerabilities that allow our attacks is the execution of untrusted JS with full access to the DOM. JS has complete read and write access to the DOM, and the submit events can be hooked. By reading data from the DOM the parasite can read email communication, e.g., from Gmail, or account numbers, e.g., on crypto exchanges, or to read the financial status in online banking. Encryption of the network traffic does not prevent the attack since the parasite can read the data directly from the input fields and then transfer it to the attacker via C&C channel. If the user is logged in, a corresponding fake login screen is presented. By manipulating the DOM the attacker can manipulate bank transfer details in online banking.

Preventing such attacks on the Browser level is hard, because the parasite utilises only standardised JS functions. The most promising way is to limit the communication between the attacker and the parasites by enforcing a strong CSP, Sub-Resource-Integrity and a fresh load of the main HTML file. In this stage of the attack the encryption of the network traffic does not help, because the data is read directly from the DOM and then transferred to the attacker via the Command & Control methods described in Section VI-C.

The defence to prevent 'two-factor authentication' attacks should require the user to confirm the transaction details on a second device. So in addition to the one-time password (OTP) there must be implemented an out-of-band transaction detail confirmation.

The vulnerabilities that allow advanced attacks, such as phishing, are the same as previously listed. Security-critical applications like web mail should have all the security measures on DNS and HTTP level enabled. Besides CSP and sub-resource-integrity, HSTS should be enabled, because it blocks the attack by enforcing HTTPS. The parasites can also execute side channel attacks against hardware. The defences to prevent such attacks are specific to the low-level systems, the parasites are used only to execute the corresponding JS based exploit code. In addition to attacking the browser and the OS of the victim, we implemented functionality to find other hosts and propagate to them. The parasite uses WebRTC to find the internal IP of the client and runs reconnaissance to find hosts via WebSockets. We fingerprint found hosts by including 'img' tags and stylesheets into the DOM, listed to onload events. Once a device is identified, the exploit starts via JS. To execute attacks against victim OS and network the victim has to open any site with manipulated files in the cache.

VIII. RECOMMENDATIONS FOR COUNTERMEASURES

We recommend to disable caching of scripts to ensure that a fresh copy is loaded every time - we implemented this by adding a random query string to each request. The used alphabet in these files is allows to compress the file a lot.

Another defence is cache partitioning which browser vendors started deploying, but studies show that it is inefficient, [11]. We also recommend that web servers support CSP, which prevents resources from being integrated by third parties. Our measurement of 15K-top Alexa domains shows that CSP is implemented in only 4.33% of the pages, and only approximately 4.7% actually supply CSP rules, from which 15.3% where using a deprecated configuration; see CSP statistics in Figure 5. Lastly, not well configured headers are supplied in those configurations as well as for example '`connect-src *;`', which simply allows every `connect-src` (and therefore also WebSockets without restriction). Out of 160 times

		Name	Targets	Exploit	Requirements
Victim Browser	C	Steal Login Data	Social networks, web mail, online banking, crypto-exchanges	Use JS access to DOM & wait for events. Exfiltrate data via C&C by encoding data to JSON, and send to server with 'src' property of an 'img' tag that is added to the DOM. We implemented modules to read browser data (user agent cookies, local storage), to extract login data by hooking into login forms (e.g., Google, Facebook, online banking apps), and tested them (e.g., on Gmail, Facebook)	→ if the user is not logged in we wait till he logs in and → if the user is logged in we show him a fake login form in the DOM.
		Browser Data	Cookies, LocalStorage	Access via Browser API	no additional requirements
		Personal Browser Data	Geolocation, microphone, webcam	Access via Browser API	Authorization by an attacked domain
		Website Data	Financial status, chats, emails...	Access via DOM	no additional requirements
		Side Channels	Side channels between the browser tabs to communicate on the machine of a victim	Timing, CPU usage...	no additional requirements
	I	Circumvent Two Factor Authentication	Google Authenticator, TAN...	Exploits de-synchronisation of knowledge between server and client. Access to the DOM allows attacker to manipulate the data and interfaces the user sees. Attack is done in JS context of attacked site.	No out-of-band transaction detail confirmation is used, or is ignored by the user.
		Transaction Manipulation	Online banking, crypto exchanges	Let the user think he does his intended transaction, but in reality he will accept an evil transaction	No out-of-band transaction detail confirmation is used, or is ignored by the user.
		Send Phishing	Web mail, social networks, WhatsApp Web ...	We harvest data out of the chat app, then use the DOM to send personalized phishing to the contacts of the user as well as read previous email communication from the DOM.	The application to attack must be open (in a tab) for sending the phishing. It suffices for a browser to be open and used on different sites.
		Steal Computation Resources	Crypto-currency mining, crack hashes, distributed scraper...	We use the CPU / GPU to perform computations.	no additional requirements
		Click Jacking	Attack noninflected sites	We have complete access to the DOM, so we can run click jacking attacks	no additional requirements
		Ad Injection	Inject ads in websites the victims visit	We can target revolvers which have many website users on them. Then inject ads in websites the victims visit.[38]	no additional requirements
	A	DDoS	Attack other sites	Use web based requests (images, web sockets...) to overload servers [25]. An infected network cache, like CDN edge server can be exploited for DDoS.	no additional requirements
Victim OS	C	JS CPU Cache & Spectre	Attack the CPU cache via timing	Attacker uses timing side channels to read data in cache [23, 22]	no additional requirements
		Rowhammer	Attack the RAM	Exploits charges leak of memory cells the exploits use privilege escalation [14]	Lack of HW techniques to prevent the rowhammer
	I	0-day on Demand	Exploit the System of the client.	The parasite loads 0-day exploits to the client and launches them.	no additional
Victim Network	I	Attack Insecure Routers and internal IoT Devices	Attack devices in the internal network of the victim	Use WebRTC and JS to scan and attack devices in the internal network of the victim (sonar.js)	no additional requirements
	A	DDoS Internal Systems	Overload devices in the targeted internal network.	Use infected clients to overload devices in the targeted internal network.[25]	no additional requirements

TABLE V
EVALUATION OF ATTACKS AGAINST POPULAR APPLICATIONS, TARGETING (C)ONFIDENTIALITY, (I)NTEGRITY AND (A)VAILABILITY.

'connect-src' being used, 17 used a wildcard here and are therefore not properly configured.

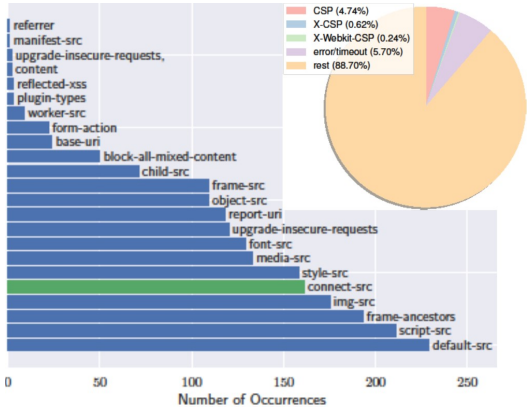

Fig. 5. CSP statistics, connect-src frequently used if CSP is supplied. Pie-chart: Used CSP version, where X-CSP and X-Webkit-CSP are deprecated.

CSP has to be configured correctly to ensure, that only trusted origins are used for remote resources like images or WebSockets. Even if all the servers are configured correctly (which is not done often [42]), not all browsers are supporting CSP and there are also bypasses for CSP, which amplifies the problem [20]. In order to draw awareness to these headers, browsers should display warnings. Another way to enforce these headers could be that major search engines use them to rank search results, as is done with HTTPS.

It is also recommended to check the integrity of the included resources via the Subresource Integrity (SRI)[4] security feature.

[4]https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity

However neither CSP nor SRI provide security during the active injection phase by eavesdropping attacker. Since the attacker spoofs the response from the server, it also controls all the headers and the delivered documents. CSP can deliver limited protection when the victim is not exposed to the attacker any more, by eliminating the persistence and the C&C.

IX. CONCLUSIONS

We develop a botnet which is based on sandboxed scripts, we call parasites, with a remote attacker which controls them. Our methodology of injecting the scripts is based on camouflaging the malicious script as appearing to originate from a genuine website, this allows us to bypass SOP restrictions. Our work demonstrates that the common belief that sandboxed scripts pose an ephemeral threat, which applies only during the visit to the malicious website, is risky. We show that the caches can be forced to store the scripts even after the victim stops visiting the website whose object was infected with a parasite, and even after the victim moves to a different network. We evaluate experimentally the fraction of objects on popular websites that can be exploited for such persistent attacks.

The main contribution of our work is to experimentally evaluate the attack surface introduced by such parasite scripts on to applications and caches.

ACKNOWLEDGEMENTS

This research has been funded by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE, and co-funded by the DFG as part of project S3 within the CRC 1119 CROSSING.

References

[1] Alharbi, F., Chang, J., Zhou, Y., Qian, F., Qian, Z., Abu-Ghazaleh, N.: Collaborative client-side dns cache poisoning attack. In: IEEE INFOCOM 2019-IEEE Conference on Computer Communications. pp. 1153–1161. IEEE (2019)

[2] appliansys: CACHEBOX on Italian Cruise (2015), https://www.appliansys.com/italian-isp-caches-on-cruises-to-save-bandwidth-and-improve-connectivity/

[3] Barth, A., Weinberger, J., Song, D.: Cross-origin javascript capability leaks: Detection, exploitation, and defense. In: USENIX security symposium. pp. 187–198 (2009)

[4] Birge-Lee, H., Sun, Y., Edmundson, A., Rexford, J., Mittal, P.: Bamboozling certificate authorities with {BGP}. In: 27th {USENIX} Security Symposium ({USENIX} Security 18). pp. 833–849 (2018)

[5] Brandt, M., Dai, T., Klein, A., Shulman, H., Waidner, M.: Domain Validation++ For MitM-Resilient PKI. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 2060–2076. ACM (2018)

[6] BuiltWith: Google Analytics Usage Statistics (2019), https://trends.builtwith.com/analytics/Google-Analytics

[7] Bursztein, E., Gourdin, B., Rydstedt, G., Boneh, D.: Bad memories (2010)

[8] Chen, S., Ross, D., Wang, Y.M.: An analysis of browser domain-isolation bugs and a light-weight transparent defense mechanism. In: Proceedings of the 14th ACM conference on Computer and communications security. pp. 2–11 (2007)

[9] Chen, W., Qian, Z.: Off-path {TCP} exploit: How wireless routers can jeopardize your secrets. In: 27th {USENIX} Security Symposium ({USENIX} Security 18). pp. 1581–1598 (2018)

[10] Citrix: Citrix NetScaler Documentation (2019), https://docs.citrix.com/en-us/netscaler/12/getting-started-with-netscaler/secure-traffic-using-ssl.html

[11] Dehling, F., Mengel, T., Iacono, L.L.: Rotten cellar: Security and privacy of the browser cache revisited. In: Nordic Conference on Secure IT Systems. pp. 20–36. Springer (2019)

[12] Eskandari, S., Leoutsarakos, A., Mursch, T., Clark, J.: A first look at browser-based cryptojacking. In: 2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW). pp. 58–66. IEEE (2018)

[13] Gilad, Y., Herzberg, A., Shulman, H.: Off-path hacking: The illusion of challenge-response authentication. IEEE Security & Privacy **12**(5), 68–77 (2013)

[14] Gruss, D., Maurice, C., Mangard, S.: Rowhammer. js: A remote software-induced fault attack in javascript. In: International conference on detection of intrusions and malware, and vulnerability assessment. pp. 300–321. Springer (2016)

[15] Herzberg, A., Shulman, H.: Security of patched dns. In: European Symposium on Research in Computer Security. pp. 271–288. Springer (2012)

[16] Herzberg, A., Shulman, H.: Fragmentation Considered Poisonous: or one-domain-to-rule-them-all.org. In: IEEE CNS 2013. The Conference on Communications and Network Security, Washington, D.C., U.S. IEEE (October 2013)

[17] Herzberg, A., Shulman, H.: Socket overloading for fun and cache-poisoning. In: Proceedings of the 29th Annual Computer Security Applications Conference. pp. 189–198. ACM (2013)

[18] Jia, Y., Chua, Z.L., Hu, H., Chen, S., Saxena, P., Liang, Z.: "the web/local" boundary is fuzzy: A security study of chrome's process-based sandboxing. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 791–804 (2016)

[19] Klein, A., Shulman, H., Waidner, M.: Internet-wide study of dns cache injections. In: IEEE INFOCOM 2017-IEEE Conference on Computer Communications. pp. 1–9. IEEE (2017)

[20] Magazinius, J., Rios, B.K., Sabelfeld, A.: Polyglots: crossing origins by crossing formats. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. pp. 753–764 (2013)

[21] Man, K., Qian, Z., Wang, Z., Zheng, X., Huang, Y., Duan, H.: Dns cache poisoning attack reloaded: Revolutions with side channels. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. pp. 1337–1350 (2020)

[22] Noack, L., Reichert, T.: Exploiting speculative execution (spectre) via javascript. Advanced Microkernel Operating Systems p. 11 (2018)

[23] Oren, Y., Kemerlis, V.P., Sethumadhavan, S., Keromytis, A.D.: The spy in the sandbox: Practical cache attacks in javascript and their implications. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. pp. 1406–1418 (2015)

[24] Papadopoulos, P., Ilia, P., Polychronakis, M., Markatos, E.P., Ioannidis, S., Vasiliadis, G.: Master of web puppets: Abusing web browsers for persistent and stealthy computation. arXiv preprint arXiv:1810.00464 (2018)

[25] Pellegrino, G., Rossow, C., Ryba, F.J., Schmidt, T.C., Wählisch, M.: Cashing out the great cannon? on browser-based ddos attacks and economics. In: 9th {USENIX} Workshop on Offensive Technologies ({WOOT} 15) (2015)

[26] Qian, F., Quah, K.S., Huang, J., Erman, J., Gerber, A., Mao, Z., Sen, S., Spatscheck, O.: Web caching on smartphones: ideal vs. reality. In: Proceedings of the 10th international conference on Mobile systems, applications, and services. pp. 127–140. ACM (2012)

[27] Qian, Z., Mao, Z.M.: Off-path tcp sequence number inference attack-how firewall middleboxes reduce security. In: 2012 IEEE Symposium on Security and Privacy. pp. 347–361. IEEE (2012)

[28] Ramanan, B.A., Drabeck, L.M., Haner, M., Nithi, N., Klein, T.E., Sawkar, C.: Cacheability analysis of http traffic in an operational lte network. In: 2013 Wireless Telecommunications Symposium (WTS). pp. 1–8. IEEE (2013)

[29] Rogowski, R., Morton, M., Li, F., Monrose, F., Snow, K.Z., Polychronakis, M.: Revisiting browser security in the modern era: New data-only attacks and defenses. In: 2017 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 366–381. IEEE (2017)

[30] Ruderman, J.: Same origin policy for javascript. Mozilla Developer Network, https://develope r. mozilla. org/ko/docs/Web/Security/Same-origin_po licy (accessed April 13, 2017) (2009)

[31] Rula, J.P., Newman, J., Bustamante, F.E., Kakhki, A.M., Choffnes, D.: Mile high wifi: A first look at in-flight internet connectivity. In: Proceedings of the 2018 World Wide Web Conference. pp. 1449–1458. International World Wide Web Conferences Steering Committee (2018)

[32] Services, A.C.: Flight Airworthiness Support Technology (2002), https://www.airbus.com/content/dam/corporate-topics/publications/fast/FAST30.pdf

[33] Shulman, H., Waidner, M.: Fragmentation considered leaking: port inference for dns poisoning. In: International Conference on Applied Cryptography and Network Security. pp. 531–548. Springer (2014)

[34] Shulman, H., Waidner, M.: Towards security of internet naming infrastructure. In: European Symposium on Research in Computer Security. pp. 3–22. Springer (2015)

[35] Shusterman, A., Kang, L., Haskal, Y., Meltser, Y., Mittal, P., Oren, Y., Yarom, Y.: Robust website fingerprinting through the cache occupancy channel. In: 28th {USENIX} Security Symposium ({USENIX} Security 19). pp. 639–656 (2019)

[36] Stamm, S., Sterne, B., Markham, G.: Reining in the web with content security policy. In: Proceedings of the 19th international conference on World wide web. pp. 921–930 (2010)

[37] Team, S.R., et al.: Emotet exposed: looking inside highly destructive malware. Network Security **2019**(6), 6–11 (2019)

[38] Thomas, K., Bursztein, E., Grier, C., Ho, G., Jagpal, N., Kapravelos, A., McCoy, D., Nappa, A., Paxson, V., Pearce, P., et al.: Ad injection at scale: Assessing deceptive advertisement modifications. In: 2015 IEEE Symposium on Security and Privacy. pp. 151–167. IEEE (2015)

[39] Vallentin, M., Ben-David, Y.: Persistent browser cache poisoning (2010)

[40] Van Kesteren, A., et al.: Cross-origin resource sharing. W3C Working Draft WD-cors-20100727, latest version available at¡ http://www. w3. org/TR/cors (2010)

[41] Peter Van de Venne Director IT, S.: Data-hungry applications via the "limited-capability" satellite network (2014), https://thedigitalship.com/conferences/presentations/2014rotterdam/08_Peter_Van_de_Venne-Director_of_IT-Spliethoff.pdf

[42] Weichselbaum, L., Spagnuolo, M., Lekies, S., Janc, A.: Csp is dead, long live csp! on the insecurity of whitelists and the future of content security policy. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 1376–1387 (2016)

[43] Zhang, K., Leng, S., He, Y., Maharjan, S., Zhang, Y.: Cooperative content caching in 5g networks with mobile edge computing. IEEE Wireless Communications **25**(3), 80–87 (2018)

[44] Zheng, X., Lu, C., Peng, J., Yang, Q., Zhou, D., Liu, B., Man, K., Hao, S., Duan, H., Qian, Z.: Poison over troubled forwarders: A cache poisoning attack targeting {DNS} forwarding devices. In: 29th {USENIX} Security Symposium ({USENIX} Security 20). pp. 577–593 (2020)