The zombies strike back: Towards client-side BeEFdetection

Maxim Chernyshev
Edith Cowan University, m.chernyshev@ecu.edu.au

Peter Hannay
Edith Cowan University, p.hannay@ecu.edu.au

DOI: 10.4225/75/57b3de3dfb87a

12th Australian Digital Forensics Conference. Held on the 1-3 December, 2014 at Edith Cowan University, Joondalup Campus, Perth, Western Australia.
This Conference Proceeding is posted at Research Online.
https://ro.ecu.edu.au/adf/133

THE ZOMBIES STRIKE BACK: TOWARDS CLIENT-SIDE BEEF DETECTION

Maxim Chernyshev[1], Peter Hannay[2]
School of Computer and Security Science[1, 2], Security Research Institute[2]
Edith Cowan University[1, 2], Perth, Australia
m.chernyshev@ecu.edu.au, p.hannay@ecu.edu.au

Abstract

A web browser is an application that comes bundled with every consumer operating system, including both desktop and mobile platforms. A modern web browser is complex software that has access to system-level features, includes various plugins and requires the availability of an Internet connection. Like any multifaceted software products, web browsers are prone to numerous vulnerabilities. Exploitation of these vulnerabilities can result in destructive consequences ranging from identity theft to network infrastructure damage. BeEF, the Browser Exploitation Framework, allows taking advantage of these vulnerabilities to launch a diverse range of readily available attacks from within the browser context. Existing defensive approaches aimed at hardening network perimeters and detecting common threats based on traffic analysis have not been found successful in the context of BeEF detection. This paper presents a proof-of-concept approach to BeEF detection in its own operating environment – the web browser – based on global context monitoring, abstract syntax tree fingerprinting and real-time network traffic analysis.

Keywords
Web browser, Google Chrome, JavaScript malware, BeEF, abstract syntax tree fingerprinting, network traffic analysis

INTRODUCTION

JavaScript is an integral part of the modern web as it enables the delivery of highly interactive web applications and is supported by all contemporary browsers. However, it can also be abused to attack client browsers. There are a number of different JavaScript-based threats, such as heap-spraying code injection attacks, cross-site scripting (XSS) and drive-by downloads (Marco Cova, Kruegel, & Vigna, 2010; Oriyano & Shimonski, 2012; Ratanaworabhan, Livshits, & Zorn, 2009). These threats are made possible because browsers are commonly configured to execute inherently untrusted code.

The Browser Exploitation Framework (BeEF) leverages untrusted code execution to provide the functionality to exploit browsers with the aim of supporting penetration testing. The ability to detect the behaviour exhibited by BeEF is important to identify client-based attacks as well as potentially hostile code. In this paper we will investigate some of the behavioural traits of BeEF and present a proof-of-concept approach aimed at its detection within the browser.

EXPLOITING THE BROWSER

Man-in-the-Browser Attacks

Man-in-the-Browser (MitB) attack operates at the Application Layer and targets the client browser (Alcorn, Frichot, & Orru, 2014, p. 60). While Man-in-the-Middle (MitM) attacks usually require a compromised or inserted hardware component to facilitate the interception, MitB attacks are software-driven (Dougan & Curran, 2012, p. 30). The general capabilities of MitB include the ability to steal data, modify client requests and server responses in real time, selective targeting and two-way communication to a command and control (C&C) server (Dougan & Curran, 2012, pp. 31-33).

Various kinds of specialised banking malware, such as Zeus, SpyEye and Bugat, leverage these capabilities. Specifically, these types of malware support transparent injection of malicious content into the original HTTP responses with the aim to trick the user into supplying their banking credentials (Dougan & Curran, 2012, p. 35; IOActive, 2012; Silva, Silva, Pinto, & Salles, 2013, p. 397; Sood, Enbody, & Bansal, 2013, p. 445). For instance, the *Web Injects* module of SpyEye as well as the *httpinjects* section of the Bugat bot configuration file allow defining targeted injection and replacement rules based on the URL, request method and provided HTML elements (StopMalvertising.com, 2014).

Browser Exploitation Framework (BeEF)

While certain types of malware infect the underlying software components and libraries to gain access to the relevant system calls, MitB capabilities can be realised purely in the context of the client browser. This approach is adopted by BeEF – an open-source project that provides a convenient mechanism to perform browser-focused penetration tests (Alcorn, 2014). The inception of BeEF dates back to 2005, when its first public release was inspired by research into advanced XSS and persistent bidirectional victim-to-attacker communication (Orru, 2011; Rager, 2005). BeEF also builds upon research into inter-protocol communication and exploitation (Alcorn, 2007).

The primary purpose of the framework is to "assess the actual security posture of a target environment by using client-side attack vectors" (Alcorn, 2014). This notion assumes a traditional approach to network security that focuses on perimeter defence and makes use of firewalls to filter packets on numerous ports but those commonly used by HTTP (Alcorn et al., 2014, p. 17). The authors of BeEF suggest that web browser attack surface is "large and ever-increasing", due to its continuously expanding feature set (Alcorn et al., 2014, pp. 18-20). BeEF can be used to target this surface in order to bypass traditional external perimeter defence and gain access to internal targets that would otherwise be inaccessible.

In practice, the usage of BeEF follows a prescribed browser hacking methodology, which consists of three high-level phases - initiating, retaining and attacking (Alcorn et al., 2014, p. 23). During the initiation phase, the browser encounters and executes the *hook* script, which contains a set of instructions required to initiate the communication channel between the browser and the C&C server. The browser can be tricked into executing this script using social engineering, XSS attacks, compromised web applications, advertising networks and MitM attacks (Alcorn et al., 2014, pp. 32-72). The retaining phase is concerned with establishing an active and persistent communication channel to enable the attacking phase. BeEF relies on HTTP to facilitate the communication via XMLHttpRequest (XHR) polling or HTML5 WebSockets. The latter offers better performance through bidirectional communication as well as reduced detection as firewalls are generally unfamiliar with this protocol (Erkkilä, 2012). Channel persistence is achieved via means of full-window frame overlays, browser events, pop-under windows and various MitB techniques. The attacking phase involves the delivery of JavaScript payloads to the browser, which executes them and communicates the result back to the C&C server. BeEF incorporates a collection of commands that can target different entities ranging from user credentials to internal network resources. The framework provides a diverse range of features including but not limited to:

- Browser fingerprinting
- Network enumeration and inter-protocol exploitation
- Key logging to extract sensitive data, such as credentials and personal information
- Ability to use the victim browser as the proxy
- Numerous social engineering modules, such as spoofed LastPass and Evernote login dialogs
- Chrome extension exploitation
- Integration with the *Metasploit* penetration-testing framework
- Client-side detection evasion via minification and obfuscation

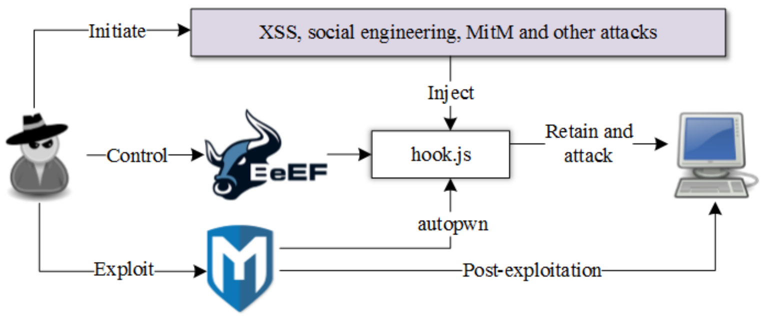


Figure 1. BeEF and Metasploit Attack Scenario

Figure 1 depicts a high-level overview of a client attack using BeEF. The browser that has executed the hook script is referred to as a *zombie* that has been *hooked*. The ability of BeEF to integrate with the *Metasploit* penetration-testing framework deserves a separate mention, as it significantly expands its capabilities. Leveraging this integration, one could take advantage of the *Metasploit* browser *autopwn* module to attempt automated browser exploitation to obtain a reverse shell on the target machine. Other tools that leverage similar concepts exist - such as Waldo by Shekyan (2012) and XSSChef by Kotowicz (2012), which focuses on Chrome browser extension exploitation.

RELATED WORK

As this study is concerned with client-side detection, we examined some of the existing techniques used to detect and protect against JavaScript malware. The taxonomy of mitigation approaches was previously proposed by Yin (2013, p. 17) and is presented in Figure 2. Based on this taxonomy, we classify the detection approaches into three types of analysis – static, dynamic and hybrid. Whereas static analysis examines the source code of a computer program statically without execution, dynamic analysis is concerned with studying the behaviour of a running program (Chess & McGraw, 2004, p. 32). A hybrid approach uses a combination of the two techniques to compensate for the shortcomings of each.

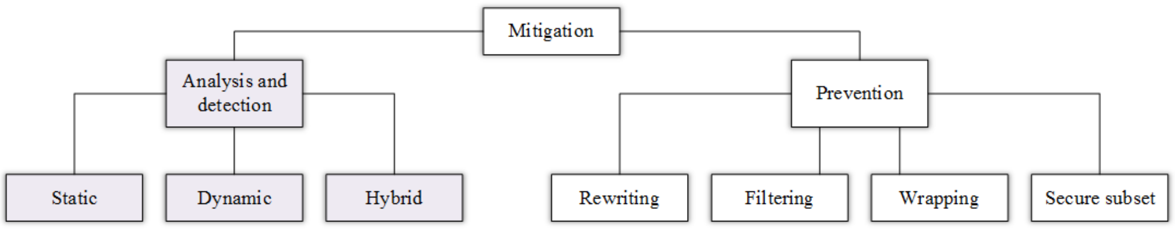


Figure 2. Taxonomy of Mitigation Approaches Against JavaScript-based Threats (adapted from Yin (2013, p. 17))

Static analysis can be used to detect specific function invocations that are present in the obfuscated code – an evasion technique often adopted by the creators of malicious scripts (Xu, Zhang, & Zhu, 2013). Another static technique involves the construction of an Abstract Syntax Tree (AST) that uses nodes to represent language constructs. This is a popular technique that can be used to identify syntax and structural elements commonly present in JavaScript malware and subsequently detect other scripts that incorporate these elements (Blanc, Miyamoto, Akiyama, & Kadobayashi, 2012; Curtsinger, Livshits, Zorn, & Seifert, 2011). In addition to malware detection, static analysis techniques can be used for code cloning and plagiarism detection (Beth, 2014; Nilsson, 2012; Roy, Cordy, & Koschke, 2009).

Dynamic analysis is usually performed in an emulated environment, such as a client honeypot or a honeyclinet, script execution sandbox or an instrumented browser (Yin, 2013, pp. 19-20). This technique has been used to create a custom sandbox that offers complete JavaScript API support where script calls are examined before being allowed access to the original global context (Terrace, Beard, & Katta, 2012). Dynamic analysis can also be implemented using an emulated web-browsing environment such as *HtmlUnit* that mimics multiple platform configurations, as JavaScript malware commonly relies on client fingerprinting to accomplish platform-specific exploitation (Marco Cova et al., 2010). Furthermore, browser emulation can be utilised to perform large-scale automated collection of malware to record infection trails for subsequent replay and in-depth analysis (Chen, Gu, Zhuge, Nazario, & Han, 2011). Client honeypots providing varying degree of interactivity can also be used to emulate common vulnerabilities to provide insight into practical client-side attack techniques (Nazario, 2009; Seifert & Steenson, 2006).

Hybrid analysis requires access to both the source code as well as the runtime execution context. For example, a runtime component can be used to intercept specific function invocations to perform the subsequent argument analysis statically (Curtsinger et al., 2011). In addition, monitoring the runtime behaviour of selected HTML tags and their attributes can be adopted to identify known malicious behaviours and inform the end-user in real time (Kishore, Mallesh, Jyostna, Eswari, & Sarma, 2014).

However, the client-side hook component of BeEF does not necessarily exhibit malicious traits. We analysed both the plain text and obfuscated versions of the client hook script using *Wepawet* and *Jsunpack*, both of which recognised the script as being benign (M Cova, 2013; Hartstein, 2009). Furthermore, we determined that detection and protection mechanisms utilised by the *JSGuard* Firefox add-on by Kishore et al. (2014) were ineffective against BeEF. These findings lead us to believe that a different approach would be required to facilitate its detection.

BEEF DETECTION

To identify detectable characteristics, we examined the architecture and behavioural patterns of BeEF. For the purposes of this paper, we assume that the initiation phase is complete and that the client browser has executed the hook script. Based on our analysis, we identified a common behavioural pattern that consists of the initial fingerprinting activity, followed by one or more commands aimed at ensuring persistence, which, in turn, are followed by context-specific attack payloads. Immediately upon the execution, the script accesses a number of specific properties to fingerprint the browser and the underlying operating platform. This information is sent back to the C&C server, which registers the browser as an active zombie.

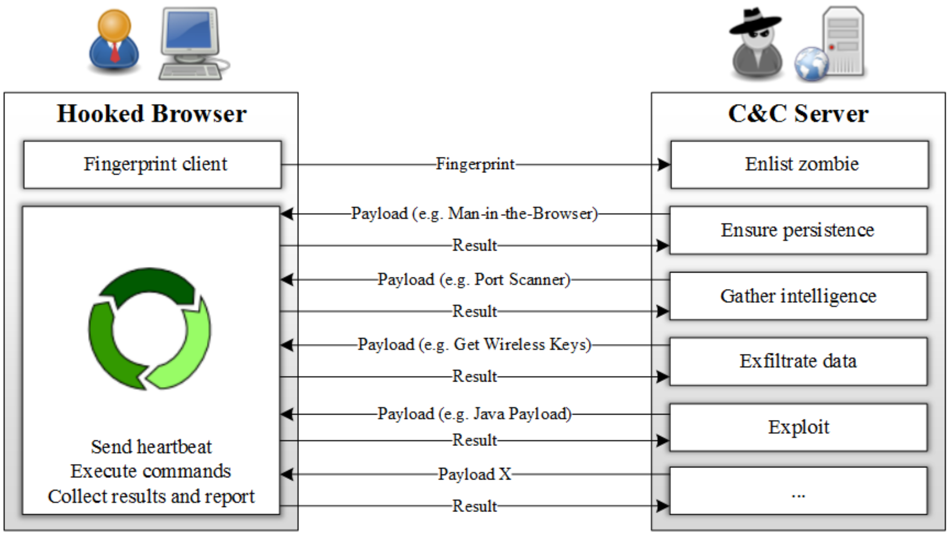
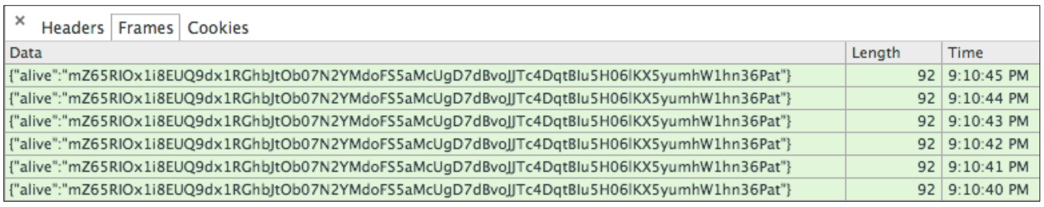


Figure 3. Representative BeEF Behavioural Pattern

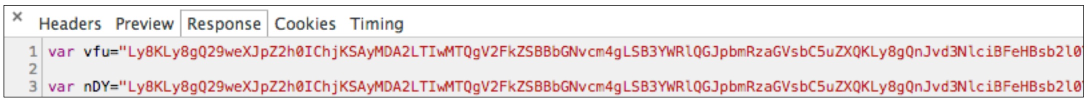
From this point on, the hook script maintains the channel by issuing heartbeat requests at predefined intervals. However, none of the necessarily malicious behaviour has been exhibited so far. For instance, XHR polling can be used as a legitimate technique to achieve near real-time communication for a rich Internet application (RIA). An example of heartbeat traffic is presented in Figure 4.



Data	Length	Time
{"alive":"mZ65RIOx1i8EUQ9dx1RGhbJtOb07N2YMdoFS5aMcUgD7dBvoJJTc4DqtBlu5H06lKX5yumhW1hn36Pat"}	92	9:10:45 PM
{"alive":"mZ65RIOx1i8EUQ9dx1RGhbJtOb07N2YMdoFS5aMcUgD7dBvoJJTc4DqtBlu5H06lKX5yumhW1hn36Pat"}	92	9:10:44 PM
{"alive":"mZ65RIOx1i8EUQ9dx1RGhbJtOb07N2YMdoFS5aMcUgD7dBvoJJTc4DqtBlu5H06lKX5yumhW1hn36Pat"}	92	9:10:43 PM
{"alive":"mZ65RIOx1i8EUQ9dx1RGhbJtOb07N2YMdoFS5aMcUgD7dBvoJJTc4DqtBlu5H06lKX5yumhW1hn36Pat"}	92	9:10:42 PM
{"alive":"mZ65RIOx1i8EUQ9dx1RGhbJtOb07N2YMdoFS5aMcUgD7dBvoJJTc4DqtBlu5H06lKX5yumhW1hn36Pat"}	92	9:10:41 PM
{"alive":"mZ65RIOx1i8EUQ9dx1RGhbJtOb07N2YMdoFS5aMcUgD7dBvoJJTc4DqtBlu5H06lKX5yumhW1hn36Pat"}	92	9:10:40 PM

Figure 4. A Sample Of C&C Heartbeat Traffic Sent Via WebSocket

To exploit the client, an attacker can choose from a plethora of commands logically categorised into modules, such as browser, network, persistence and social engineering. Command invocation involves the delivery and execution of JavaScript payloads that often have configurable parameters to tailor the attack to the particular environment. Once executed, the payload is queued and delivered to the client. Moreover, multiple payloads can be delivered as part of the same C&C response if multiple commands have been queued within the configured polling interval.

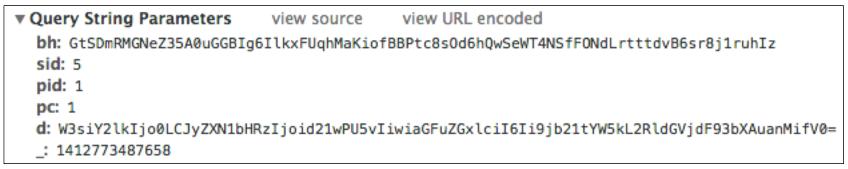


```
var vfu="Ly8KLy8gQ29weXJpZ2h0IChjKSAyMDA2LTIwMTQgV2FkZSBBbGNvcm4gLSB3YWRlQGJpbmRzaGVsbC5uZXQKLy8gQnJvd3NlciBFeHBsb2l0

var nDY="Ly8KLy8gQ29weXJpZ2h0IChjKSAyMDA2LTIwMTQgV2FkZSBBbGNvcm4gLSB3YWRlQGJpbmRzaGVsbC5uZXQKLy8gQnJvd3NlciBFeHBsb2l0
```

Figure 5. Payload Delivery as Obfuscated JavaScript Via XHR Polling

An example of a multi-command payload is presented in Figure 5. As an evasion technique, the payloads can be obfuscated and an additional evasion technique is used to conceal the execution call. Upon the execution, the results are sent back to the C&C server. When using an XHR-based communication channel, results can be delivered in multiple base64-encoded chunks of pre-defined size as an additional evasion mechanism. An example of command execution result parameter set is presented in Figure 6.



```
bh: GtSDmRMGNeZ35A0uGGBIg6IlkxFUqhMaKiofBBPtc8sOd6hQwSeWT4NSfFONdLrtttdvB6sr8j1ruhIz
sid: 5
pid: 1
pc: 1
d: W3siY2lkIjo0LCJyZXN1bHRzIjoid21wPU5vIiwiaGFuZGxlciI6Ii9jb21tYW5kL2RldGVjdF93bXAuanMifV0=
_: 1412773487658
```

Figure 6. Payload Execution Results Delivery Via XHR Polling

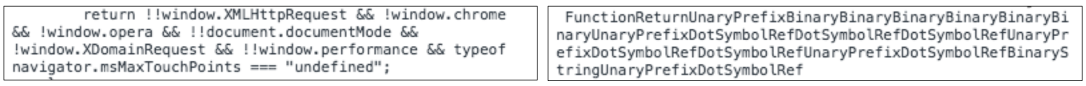
To speed up the exploitation process, command execution can be automated. Using the BeEF Injection Framework by SpiderLabs (2014) an attacker could specify a set of commands and corresponding parameters to be executed automatically upon the browser becoming hooked.

Browser Fingerprinting Activity

The exhibited fingerprinting activity appears to rely on more than 2,000 property access invocations. In contrast, the original project by the Electronic Frontier Foundation (EFF) achieves this goal using less than 600 property access calls (Eckersley, 2011). Another library called *fingerprintjs* performs this task using less than 10 calls (Vasilyev, 2014). While high-volume fingerprinting activity is not an explicit indication of malware, it could be used to suggest potential BeEF presence, especially when coupled with other indicators.

Global Namespace Object

Our inspection of the client-side hook script revealed that it registered a custom object in the global window namespace. The evasion techniques employed by BeEF can be used to randomise its name as well as the names of any other variables referenced within, which can be used to bypass static filters. However, previous work indicates that a combination of analysis techniques including AST-based fingerprinting can be adapted to aid with detection under these circumstances.



```
        return !!window.XMLHttpRequest && !window.chrome
&& !window.opera && !!document.documentMode &&
!window.XDomainRequest && !!window.performance && typeof
navigator.msMaxTouchPoints === "undefined";
```

```
 FunctionReturnUnaryPrefixBinaryBinaryBinaryBinaryBinaryBi
naryUnaryPrefixDotSymbolRefDotSymbolRefDotSymbolRefUnaryPr
efixDotSymbolRefDotSymbolRefUnaryPrefixDotSymbolRefBinaryS
tringUnaryPrefixDotSymbolRef
```

Figure 7. Sample Function Source (left) and Corresponding Concatenated String AST Representation (right) for Hash Value "e7f892feca616bf59b59949afe8e3264a721f272"

While runtime function decompilation in JavaScript is non-standard and implementation-dependent, WebKit-based browsers such as Google Chrome provide the ability to access the source code of user-defined functions (Zaytsev, 2014). Subsequently, we used a scriptable environment based on *PhantomJS* and *CasperJS* by Hidayat (2014); Perriault (2013) to construct and compute the hashes of the ASTs of all methods available in the global BeEF object version 0.4.3.1 to 0.4.5.0. A sample function source and its corresponding AST string are presented in Figure 7. The version range represents more than two years of development and includes most of the Ruby-based versions publically available at the time of writing. As shown on Table 1, we determined that 95 method hashes were present in all of the analysed versions and 80% of versions had unique version-specific methods.

Table 1. Method AST Hash Statistics for Analysed BeEF Versions

Version	*Common*	*Non-common*	*Total*	*Unique*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
0.4.3.1	95	41	136	5	10	1	5	3	0	2	1	2	1
0.4.3.2	95	41	136	2	10	3	5	3	0	2	1	2	2
0.4.3.3	95	43	138	4	1	3	5	3	2	6	1	2	2
0.4.3.4	95	45	140	2	1	3	5	3	2	7	2	2	2
0.4.3.5	95	47	142	4	0	1	2	3	3	7	2	2	2
0.4.3.6	95	51	146	2	1	2	3	2	3	8	2	3	2
0.4.3.7	95	53	148	3	1	1	3	2	3	8	2	3	2
0.4.3.8	95	54	149	2	1	1	3	2	3	7	2	3	2
0.4.3.9	95	56	151	3	2	0	1	2	1	7	1	3	3
0.4.4.1	95	59	154	3	3	0	0	2	1	3	1	1	3
0.4.4.2.1	95	67	162	4	2	0	0	0	0	5	1	1	2
0.4.4.3	95	70	165	3	0	2	0	0	0	5	0	1	6
0.4.4.4	95	75	170	0	2	2	1	1	1	4	1	9	6
0.4.4.4.1	95	75	170	0	2	2	1	1	1	4	1	9	6
0.4.4.5	95	74	169	2	0	0	2	1	1	5	1	8	6
0.4.4.6	95	77	172	2	0	1	2	1	3	5	1	8	6
0.4.4.6.1	95	78	173	0	3	1	1	2	3	5	1	8	6
0.4.4.7	95	78	173	0	3	1	6	1	3	2	1	8	6
0.4.4.8	95	79	174	4	1	3	5	1	2	2	1	8	5
0.4.4.9	95	80	175	4	8	3	5	1	2	2	1	8	5
0.4.5.0	95	85	180	11	7	3	5	1	2	2	0	8	5

Subsequently, having access to the global namespace, we could analyse non-native global objects to identify matches against the derived collection of AST hash-based fingerprints.

Heartbeat Traffic

According to GitHub (2012), network communication obfuscation through endpoint URL randomisation is listed on the current roadmap, but it has not been implemented yet. This means that the same C&C server endpoint is utilised for heartbeat traffic. XHR-based heartbeat messages follow the same format with the only difference being the value of the "_" parameter that is updated to reflect the current timestamp. In the case of WebSockets, the socket is used to deliver the heartbeat messages as well as commands and execution results. WebSocket-based heartbeat messages are identical. The frequency of these messages is determined by the configured polling interval, which does not change while the C&C server is running. Assuming BeEF heartbeat traffic is characterised by messages of identical length being delivered at regular intervals to the same endpoint, we could utilise timing and length-based analysis to detect this behaviour (Wright, Monrose, & Masson, 2006).

Command Payloads

Regardless of whether the built-in evasion mechanisms are used or not, the delivered payloads are represented as syntactically valid JavaScript, as execution involves running it verbatim. Until a more evasive technique is introduced, XHR response and socket message payload analysis could be used to identify JavaScript payloads as one of the detectable characteristics.

Practical Detection Approach

Our detection approach is based on observing the previously described characteristics in the context of the client browser. To achieve this goal, we implemented a custom Google Chrome extension. The latest extension development framework addresses the previous limitations described by Heiderich, Frosch, and Holz (2011, p. 12). The architecture of the BeEF detector extension is presented in Figure 8. The content script is injected into the limited-access context before any other scripts. Subsequently, the content script is able to inject a custom window script into the global context to complete the necessary environment preparation in time. Specifically, it ensures that fingerprinting activity as well as XHR and WebSocket-based traffic can be intercepted and that the source of user-defined functions is accessible in its original form.

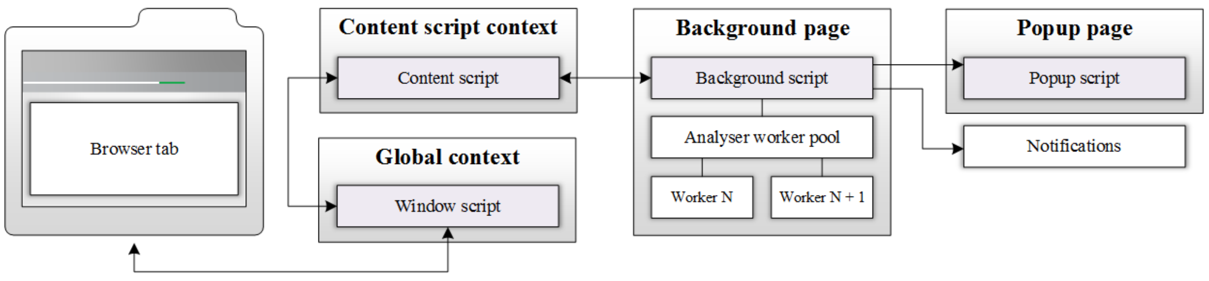
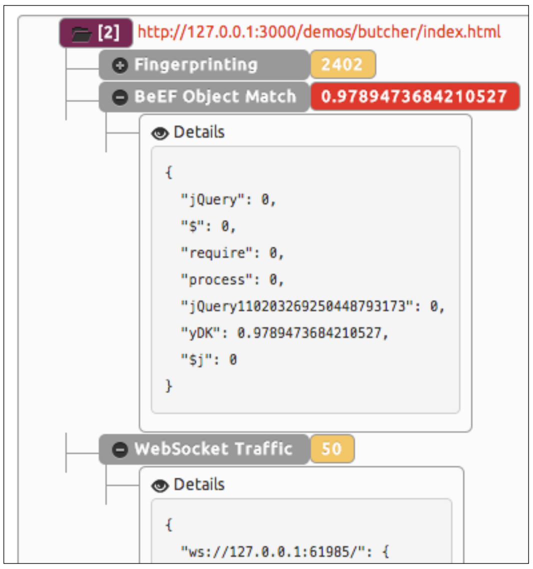


Figure 8. BeEF Detector Chrome Extension Architecture

The window script uses custom events to communicate with the content script, which uses native extension messaging to communicate with the background script that runs in the background persistently. This means that any resource-intensive processing such as AST hash computations can be offloaded to the background page to minimise the impact on the browsing experience. The background page uses a worker pool to parallelise the analysis tasks.



```
{
  "jQuery": 0,
  "$": 0,
  "require": 0,
  "process": 0,
  "jQuery110203269250448793173": 0,
  "yDK": 0.9789473684210527,
  "$j": 0
}
```

```
{
  "ws://127.0.0.1:61985/": {
```

Figure 9. Real-Time Detection Status Feedback Using a Popup Page

The background page also uses native extension messaging to communicate with a separate popup page that is used to provide real-time feedback on detectable characteristics for all browser tabs, as shown in Figure 9. Furthermore, the Chrome notifications API is used to issue user warnings on significant events such as high probability BeEF object match or JavaScript payload detection.

DISCUSSION

The proof-of-concept extension was implemented and tested against the analysed versions of BeEF. The extension was able to distinguish excessive fingerprinting behaviour, identify the BeEF object in the global window namespace, as well as detect heartbeat traffic and JavaScript payloads within seconds of the hook script being executed. To achieve a wider practical utilisation, the finalised extension could potentially be bundled as part of the enterprise Managed Operating Environment (MOE) or offered via the Chrome Web Store.

As part of testing, we also discovered a weakness in the handling of the incoming data by the C&C endpoint when using WebSockets. The protocol suggests that the connection should be closed when invalid data is received (Erkkilä, 2012). However, sending an arbitrary message renders the C&C server unresponsive requiring a subsequent restart. Consideration should be given to the incorporation of additional evasion techniques and defensive mechanisms to decrease the likelihood of BeEF becoming a target of such counterattack. For instance, caching commonly accessed properties could reduce the visible fingerprinting activity and anonymous functions could be utilised to conceal the BeEF object (Heiderich et al., 2011, p. 12).

While our initial tests found the described approach to be effective, it is not without limitations. Firstly, the extension is only compatible with the Chrome browser and additional research is required to determine the feasibility of porting it to other browsers. Secondly, the detection requires the window script to be injected and executed before any other scripts and, therefore, timing issues could impair the detection ability. Thirdly, the approach relies on a signature database that will need to be kept up-to-date to account for new releases. Finally, the approach has not been verified against the attacks in the wild and it may be possible that there are heavily modified versions that do not exhibit the analysed characteristics.

CONCLUSION

Client-side attacks based on JavaScript abuse continue to be of concern, reinforcing the need for improved browser security and specialised protection mechanisms. In the case of BeEF, while the client-side component is not necessarily inherently malicious, it does facilitate means of launching specialised attacks in a controlled and possibly automated way. In this paper we focused on the identification of detectable behavioural characteristics of BeEF and described a proof-of-concept detection approach that was verified against a number of recent publically available versions. The initial findings suggest its potential suitability, but also highlight a number of important limitations. This study is the first step towards client-side BeEF detection and future research should aim at conducting a wider evaluation and further refinement and optimisation of the described proof-of-concept.

REFERENCES

Alcorn, W. (2007). *InterProtocol Exploitation*.

Alcorn, W. (2014). BeEF - The Browser Exploitation Framework Project. Retrieved June 24, 2014, from http://beefproject.com/

Alcorn, W., Frichot, C., & Orru, M. (2014). *The Browser Hacker's Handbook* Retrieved from http://ECU.eblib.com.au/patron/FullRecord.aspx?p=1641459

Beth, B. (2014). A Comparison of Similarity Techniques for Detecting Source Code Plagiarism.

Blanc, G., Miyamoto, D., Akiyama, M., & Kadobayashi, Y. (2012). *Characterizing Obfuscated JavaScript Using Abstract Syntax Trees: Experimenting with Malicious Scripts.* Paper presented at the Advanced Information Networking and Applications Workshops (WAINA), 2012 26th International Conference

Chen, K. Z., Gu, G., Zhuge, J., Nazario, J., & Han, X. (2011). *WebPatrol: Automated collection and replay of web-based malware scenarios.* Paper presented at the Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security.

Chess, B., & McGraw, G. (2004). Static analysis for security. *IEEE Security & Privacy, 2*(6), 76-79.

Cova, M. (2013). Wepawet. from http://wepawet.cs.ucsb.edu/

Cova, M., Kruegel, C., & Vigna, G. (2010). *Detection and analysis of drive-by-download attacks and malicious JavaScript code.* Paper presented at the Proceedings of the 19th international conference on World wide web.

Curtsinger, C., Livshits, B., Zorn, B. G., & Seifert, C. (2011). *ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection.* Paper presented at the USENIX Security Symposium.

Dougan, T., & Curran, K. (2012). Man in the browser attacks. *International Journal of Ambient Computing and Intelligence (IJACI), 4*(1), 29-39.

Eckersley, P. (2011). Panopticlick. Retrieved October 6, 2014, from https://panopticlick.eff.org/index.php?action=log&js=yes

Erkkilä, J.-P. (2012). WebSocket Security Analysis. *Aalto University School of Science*, 2-3.

GitHub. (2012). BeEF Milestones. from https://github.com/beefproject/beef/wiki/Milestones

Hartstein, B. (2009). *Jsunpack: An automatic javascript unpacker.* Paper presented at the ShmooCon convention.

Heiderich, M., Frosch, T., & Holz, T. (2011). *IceShield: detection and mitigation of malicious websites with a frozen DOM.* Paper presented at the Recent Advances in Intrusion Detection.

Hidayat , A. (2014). PhantomJS. from http://phantomjs.org/

IOActive. (2012). Reversal and Analysis of the Zeus and SpyEye Banking Trojans.

Kishore, K. R., Mallesh, M., Jyostna, G., Eswari, P., & Sarma, S. S. (2014). *Browser JS Guard: Detects and defends against Malicious JavaScript injection based drive by download attacks.* Paper presented at the Applications of Digital Information and Web Technologies (ICADIWT), 2014 Fifth International Conference on the.

Kotowicz, K. (2012). XSS ChEF - Chrome extension exploitation framework. from http://blog.kotowicz.net/2012/07/xss-chef-chrome-extension-exploitation.html

Nazario, J. (2009, 2009). *PhoneyC: a virtual client honeypot.*

Nilsson, E. (2012). Abstract Syntax Tree Analysis for Plagiarism Detection.

Oriyano, S.-P., & Shimonski, R. (2012). *Client-Side Attacks and Defense*. Boston: Syngress.

Orru, M. (2011). *Ground BeEF: Cutting, devouring and digesting the legs off a browser*. Paper presented at the SecurityByte.

Perriault, N. (2013). CasperJS.

Rager, A. (2005). *Advanced Cross-Site-Scripting with Real-time Remote Attacker Control*. Avaya Labs.

Ratanaworabhan, P., Livshits, V. B., & Zorn, B. G. (2009). *NOZZLE: A Defense Against Heap-spraying Code Injection Attacks.* Paper presented at the USENIX Security Symposium.

Roy, C. K., Cordy, J. R., & Koschke, R. (2009). Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming, 74*(7), 470-495.

Seifert, C., & Steenson, R. (2006). Capture-honeypot client (capture-hpc). *pp. Available at https://projects. honeynet. org/capture–hpc*.

Shekyan, S. (2012). The Tiny Mighty Waldo. Retrieved May 26, 2014, from https://community.qualys.com/blogs/securitylabs/2012/08/03/the-tiny-mighty-waldo

Silva, S. S., Silva, R. M., Pinto, R. C., & Salles, R. M. (2013). Botnets: A survey. *Computer Networks, 57*(2), 378-403.

Sood, A. K., Enbody, R. J., & Bansal, R. (2013). Dissecting SpyEye – Understanding the design of third generation botnets. *Computer Networks, 57*(2), 436-450. doi: http://dx.doi.org/10.1016/j.comnet.2012.06.021

SpiderLabs. (2014). BeEF Injection Framework from https://github.com/SpiderLabs/beef_injection_framework

StopMalvertising.com. (2014). Analysis of Cridex / Feodo / Bugat. from http://stopmalvertising.com/malware-reports/analysis-of-cridex-feodo-bugat.html

Terrace, J., Beard, S. R., & Katta, N. P. K. (2012). JavaScript in JavaScript (js. js): Sandboxing third-party scripts. *USENIX WebApps*.

Vasilyev, V. (2014). fingerprintJS. Retrieved October 6, 2014, from http://valve.github.io/fingerprintjs/

Wright, C. V., Monrose, F., & Masson, G. M. (2006). On inferring application protocol behaviors in encrypted network traffic. *The Journal of Machine Learning Research, 7*, 2745-2769.

Xu, W., Zhang, F., & Zhu, S. (2013). *JStill: mostly static detection of obfuscated malicious JavaScript code.* Paper presented at the Proceedings of the third ACM conference on Data and application security and privacy.

Yin, Z. (2013). Evolution of client side web attacks and defences against malicious JavaScript code: a retrospective, systematization study.

Zaytsev, J. (2014). State of function decompilation in Javascript. from http://perfectionkills.com/state-of-function-decompilation-in-javascript/