A Spicy Approach to WebSockets: Enhancing Bro's WebSockets Network Analysis by Generating a Custom Protocol Parser with Spicy

Jennifer Gates
A Spicy Approach to WebSockets: Enhancing Bro’s WebSockets Network Analysis by Generating a Custom Protocol Parser with Spicy

GIAC (GCIA) Gold Certification

Author: Jennifer Gates, jenngates2004@gmail.com
Advisor: Dave Hoelzer
Accepted: September 19, 2017

Abstract

Although the Request for Comments (RFC) defining WebSockets was released in 2011, there has been little focus on using the Bro Intrusion Detection System (IDS) to analyze WebSockets traffic. However, there has been progress in exploiting the WebSockets protocol. The ability to customize and expand Bro’s capabilities to analyze new protocols is one of its chief benefits. The developers of Bro are also working on a new framework called Spicy that allows security professionals to generate new protocol parsers. This paper focuses on the development of Spicy and Bro scripts that allow visibility into WebSockets traffic. The research conducted compared the data that can be logged with existing Bro protocol analyzers to data that can be logged after writing a WebSockets protocol analyzer in Spicy. The research shows increased effectiveness in detecting malicious WebSockets traffic using Bro when the traffic is parsed with a Spicy script. Writing Bro logging scripts tailored to a particular WebSockets application further increases their effectiveness.
1. Introduction

It is challenging to calculate the number of web applications that use WebSockets, but considering that all major web browsers support the protocol (Deveria, 2017), it is safe to say there are currently more than there were when the Request for Comments (RFC) that defines WebSockets was released in 2011 (Fette & Melnikov, 2011). Some of the larger employments today include widely-used web application tools such as Slack, Trello, and ZenDesk (Socket.io, 2017).

Recent vulnerabilities discovered in WebSockets implementations illustrate the need for some visibility into the WebSockets protocol. For example, a bug was found in Slack that allows stealing a user’s private token (Rosén, 2017). Having the token allows full access to the user’s account. Another vulnerability found in the ping function of the WebSockets client/server for Node.js allows remote memory disclosure (Aboukhadieh & Buss, 2016). Although there is a need for visibility into WebSockets traffic, the Bro Intrusion Detection System (IDS) does not yet have a native WebSockets protocol analyzer to allow for direct inspection of WebSockets traffic (The Bro Project, 2017). However, Bro can be customized to provide this visibility.

1.1. Customizing Bro

The ability to customize and expand Bro’s capabilities to analyze new protocols is one of its unique benefits. To do so requires adding a protocol analyzer to the event engine so that key events within the protocol are generated. Event handlers must then be written to perform desired actions when the protocol’s events are triggered (Paxson, 1999, p. 5). These desired actions are the customizations added to Bro to expand its monitoring and detection capabilities.

Writing a native protocol parser for Bro is complicated. It requires knowledge of a programming language such as C++. Some protocols can be quite complex, but accurate analysis still requires the parser code to be efficient and robust. The process is tedious and error-prone, and yet the parser must be designed to handle network traffic in real-time. The parser must manage the edge cases of a protocol that may not be sufficiently

Author Name, email@addressm
defined, and not break when the traffic does not conform to the protocol, intentionally or unintentionally. Additionally, a poorly written parser will introduce vulnerabilities to the system employing it, which is often a security system such as the Bro IDS. (Pang, Paxson, Sommer, & Peterson, 2006, p. 1)

Bro’s current distribution includes a declarative language and compiler called BinPAC that can be used to generate a network protocol parser. The coder defines the protocol using BinPAC’s high-level meta-grammar and associated semantics which the parser generator translates into low-level code (Pang, Paxson, Sommer, & Peterson, 2006, p. 2). However, BinPAC still requires the coder to incorporate some C++ code to “steer parsing and track state” (Sommer, Amann, & Hall, 2016, p. 10). Thus, generating a new protocol parser using BinPAC requires a deeper understanding of programming and the C++ language than is typically practiced by security professionals.

1.2. Introducing Spicy

The developers of Bro and BinPAC have since introduced a new framework called Spicy for dissecting wire format data. Spicy builds on their experiences and lessons learned while developing Bro and BinPAC. The framework includes a type-based specification language, a just-in-time compiler toolchain, and an extensive application program interface (API) (Sommer, Amann, & Hall, 2016). Unlike BinPAC, Spicy’s type-based language includes semantics and syntax in a single language and therefore does not require the coder to use C++ for things like tracking state. Elements can be expressed semantically in the same high-level format as the protocol specification (Sommer, Amann, & Hall, 2016, p. 2). Although Spicy is still in the prototype phase, their findings are promising with regard to simplifying the process of implementing dissectors, reducing the number of lines of code, and increasing parsing speeds (Sommer, Amann, & Hall, 2016, p.8).

The Spicy framework comes with a Bro plugin that can be used with the current development version of Bro (v2.5.1 as of this writing) due to its support of dynamically loaded plugins. The plugin enables Bro to add Spicy dissectors at startup. At startup, the dissectors hook into the event engine and have their events sent to Bro’s scripting language just like the native protocol analyzers (The International Computer Science
Institute, 2014). This process allows security professionals to customize Bro with new protocol analyzers to expand its monitoring and detection capabilities for new protocols.

1.3. Literature Review of Existing Research

After an extensive literature review, it does not appear the narrow topic of enhancing Bro’s WebSockets network analysis capabilities by generating a custom protocol parser with Spicy has been conducted. However, there has been tangential research focusing on Bro intrusion detection, WebSockets security, and protocol parsing.

1.3.1. Research with Bro

Of the articles found describing research studying intrusion detection using Bro, none address using a Spicy generated WebSockets parser. Some of the research focused on network intrusion detection using Bro with its native protocol parsers, some looked at Bro parsing protocols with BinPAC generated parsers, and one used Spicy to generate parsers for Bro. The research that used Spicy was influential because the researchers experienced some setbacks due to instabilities in the not-yet-for-production version of Spicy. Specifically, some of their network traffic could not be analyzed because the parser consistently crashed while processing it. The researchers narrowed the issue to memory handling and reported it to the Spicy developers for resolution in later versions of Spicy. Despite the complications, the researcher notes that Spicy “looks very promising” (Udd, 2015, p. 25). His investment in providing feedback to the developers and remark on the potential of Spicy’s future usefulness provided validation for the value of future research despite possible complications due to early versions of the framework.

Within the research articles seeking to expand Bro’s capability with a custom generated protocol parser, most focus on Industrial Control Systems (ICS) or Supervisory Control and Data Acquisition (SCADA) protocols. For example, Adapting Bro into SCADA: building a specification-based intrusion detection system for the DNP3 protocol was looking for intrusions over the Distributed Network Protocol (DNP3) used between process automation systems (Lin, Slagell, Di Martino, Kalbarczyk, & Iyer, 2013). “Through the eye of the PLC” interrogated MODBUS protocol packets communicating process operations between Programmable Logic Controllers (PLC) (Hadžiosmanović,
A Spicy Approach to WebSockets: Enhancing Bro’s WebSockets Network Analysis by Generating a Custom Protocol Parser with Spicy

Sommer, Zambon, & Hartel, 2014). “Exploiting Bro for Intrusion Detection in a SCADA System” used Spicy to generate a custom parser for the IEC 60870-5-104 telecontrol transmission protocol. (Udd, Asplund, Nadjm-Tahrani, Kazemtabrizi, & Ekstedt, 2016). In contrast to WebSockets, the ICS/SCADA protocols parsed (DNP3 and IEC 60870-5-104) are well-defined regarding the fields within a packet, the data expected within a field, and the expected data’s meaning. The WebSockets protocol defines only a few fields, leaving the majority of the packet for data. This reduction in overhead leaves the formatting and handling of the application data to the WebSockets application (Fette & Melnikov, 2011).

Another similar research project titled “Incorporation of Application Layer Protocol Syntax into Anomaly Detection” used Bro with a BinPAC-generated HTTP parser. The project focused on detecting anomalies in the protocol elements present in request/response messages in HTTP. WebSockets traffic initiates within an HTTP GET request, but additional traffic uses the WebSockets protocol. The native HTTP protocol parsers and analyzers do not provide much insight into WebSockets application layer data (Düssel, Gehl, Laskov, & Rieck, 2008).

1.3.2. Research on WebSockets Security

Other research that focused on detecting web-based attacks on applications using WebSockets used methods other than network IDS for intrusion detection. For example, “Anomaly detection of web-based attacks” described anomaly detection using the web server system logs and HTTP query parameters. Although not directly relevant to parsing the WebSockets protocol with Bro and Spicy, it provides valuable background information regarding anomaly models and anomaly detection (Kruegel & Vigna, 2003). Once the parsing and protocol analysis of the WebSockets protocol is successful, anomaly detection techniques such as those in research by Düssel, Gehl, Laskov, & Rieck (2008), Kruegel & Vigna (2003) and Ingham, Somayaji, Burge, & Forrest (2017) may prove applicable. Since each WebSockets application’s data will look different, there is value in anomaly detection techniques that automate learning what normal and abnormal traffic looks like in each unique application’s payload data.
Research articles found that described WebSockets security tended to focus on either how to write a secure WebSockets application or how to hack with WebSockets. *WebSocket security analysis* (Erkkilä, 2012), “OWASP Top 10 Details About WebSocket Vulnerabilities and Mitigations - SecureLayer7” (Banawar, 2017), and *Using HTML5 WebSockets Securely* (Shema, 2013) demonstrate sound practices for securing a WebSockets application. *Talking to yourself for fun and profit* (Barth, Huang, Chen, Rescorla, & Jackson, 2011), *Hacking with WebSockets* (Shema, Shekyan, & Toukharian, 2013), *Hacking Slack using postMessage and WebSocket-reconnect to steal your precious token* (Rosén, 2017), and “Why servers should fear their clients: abusing WebSockets in browsers for DoS” (Parra & Posegga, 2015) each provide techniques for exploiting vulnerabilities in WebSockets. One example of such an exploit is a WebSockets handshake vulnerability that allows an attacker to retrieve third-party content by attempting to open a WebSockets connection to a third-party server that is not a WebSockets server (Parra & Posegga, 2015). Other articles discuss the possibility of remote shell access obtained through WebSockets and JavaScript code in the browser (Erkkilä, 2012), and exploits creating covert channels and conducting Command and Control (C2) through WebSockets (Shema, Shekyan, & Toukharian, 2013).

The blog post “OWASP Top 10 Details About WebSockets Vulnerabilities and Mitigations - SecureLayer7” discussed Cross-Site WebSockets Hijacking, sensitive information disclosure, and Denial of Service (DoS) concerns with WebSockets (Banawar, 2017), and provided a link to the OWASP methodology for Testing WebSockets (OWASP, 2016). The OWASP testing framework was the basis for the intrusion detection tests conducted in this research. The OWASP site also provided the Damn Vulnerable Web Sockets (DVWS) Project (OWASP, 2017) used as the vulnerable WebSockets application server in this research.

1.3.3. **Research on Protocol Parsing**

The article titled “Spicy” in the *Proceedings of the 32nd Annual Conference on Computer Security Applications - ACSAC ’16* (Sommer, 2016) and the Spicy project’s documentation website (The International Computer Science Institute, 2014) provided the Spicy language semantics and example protocol parsing scripts. Viewing the Bro source
A Spicy Approach to WebSockets: Enhancing Bro’s WebSockets Network Analysis by Generating a Custom Protocol Parser with Spicy

code for existing protocol parsers, such as HTTP, also provided general parsing techniques.

The WebSockets RFC detailed the protocol specifications, and source code on several GitHub repositories demonstrated parsing WebSockets in other languages such as C (Somerville, 2015), python (Katzarsky, 2015), and ruby (Fernández-Capel, 2015).

2. The Spicy Approach

This research focuses on comparing WebSockets traffic analysis data that could be provided by customizing the Bro IDS. In the first case, just the Bro native HTTP parser was used with a custom Bro script to create a WebSockets handshake log file. In the second case, a custom WebSockets protocol parser generated by Spicy was used, along with custom Bro scripts to create WebSockets log files. The first Bro script using Spicy-generated events solely duplicated the handshake data logged by the Bro script in the first case. The following Bro scripts expanded on the WebSockets data the Spicy parser provided to log more detailed WebSockets information and to detect malicious traffic. The scripts were then tested in a lab environment to compare the effectiveness of detecting malicious WebSockets traffic. The rest of section 2 describes this research in detail.

2.1. Writing the scripts

2.1.1. Bro Alone Scripts

The first part of this research involved writing scripts to see what WebSockets data could be provided with Bro alone. This information was later compared with what could be provided using Bro and a Spicy-generated WebSockets protocol parser.

A WebSockets connection begins with a handshake sequence through an HTTP connection upgrade request. The handshake involves an HTTP GET Request and Response with WebSockets specific headers. Bro natively parses HTTP traffic. However, it does not create a native log for just WebSockets handshake information, nor does it include WebSockets header information in the http.log file. Bro does provide an event called http_all_headers which will return the connection object for that traffic and a list...
of the header fields in the packet. The header list contains name/value pairs for each header field and its contents.

Using the http_all_headers event, the httpWSupgrade.bro script (full script provided in the Appendix) creates the httpWSupgrade.log file. The log file provides WebSockets handshake header information such as the host, URI, origin, location, subprotocols, and extensions, as well as connection information such as the timestamp, UID, server IP address, server port, and client IP address.

### 2.1.2. Using Spicy with Bro

Using Spicy with Bro to produce WebSockets data in Bro logs involves installing the Spicy/Bro plug-in to add support for Spicy grammars to Bro. Once the plugin is installed, the process requires four different types of scripts. These four types are the Bro signature script (*.sig), the Spicy parsing script (*.spicy), the Bro protocol analyzer definition script (*.evt), and the Bro logging script (*.bro) (Sommer, 2016).

The Bro signature script (*.sig) is part of the signature framework which functions at a low level and is often used to provide signature matching functionality similar to the Snort IDS. However, the signature file can also be used to activate a protocol analyzer via a signature instead of relying on the port the traffic is using. The file will define the signature by the IP protocol (TCP or UDP), a key identifying factor of the payload (such as a regular expression), and if it is the traffic originator or responder (optional). It then specifies which protocol analyzer to enable (The Bro Project, 2017).

The Spicy script (*.spicy) is used to automatically generate the protocol parser at runtime by the Spicy framework. The parser performs its actions on the data coming in from the wire just after Bro parses the lower layers including the TCP header. Writing this script requires in-depth knowledge of the protocol specifications to dissect each piece of information from the flow of the bits. It also requires an understanding of the Spicy scripting language but does not require C++ programming knowledge.

The Bro protocol analyzer definition script (*.evt) provides the interface between Bro and the Spicy script. It specifies the grammar file that contains the Spicy script and...
where to hook into Bro’s traffic processing. It then defines the Bro events to create based on the parsed data (The International Computer Science Institute, 2014).

The Bro logging scripts (*.bro) are where an analyst can customize what information will be logged (The Bro Project, 2017). Logs can provide protocol information about each connection, such as logging the WebSockets opcode, masking key, and payload length, or can be narrowed down to provide precise data in the WebSockets payload. Because Bro is a feature-rich scripting language, an adept analyst can be extremely accurate in what data he or she extracts from the traffic. An analyst can use this precision to look for a defined Indicator of Compromise (IOC) or aggregate the data to look for anomalies. Both methods are vital in finding malware.

### 2.1.3. WebSockets Protocol Parsing

To understand how the Spicy WebSockets parsing script was written, it is important to know the WebSockets protocol. Per the RFC (Fette & Melnikov, 2011), the WebSockets protocol starts after the TCP three-way handshake with the client issuing an HTTP GET Request. In addition to the standard GET request requirements to meet the HTTP protocol, the handshake request must include the Upgrade header field with a value of “WebSockets”, the Connection header field with a value of “Upgrade”, and the Sec-WebSocket-Version header field with a value of “13”. The handshake GET request must also contain the Sec-WebSocket-Key field with a 16-byte nonce value. The value is a random string that has been base64-encoded. The server uses the nonce in its response to prove that it is responding to a request from the client.

The handshake GET request may also contain optional HTTP header fields that could affect the WebSockets connection, such as Sec-WebSocket-Protocol, Sec-WebSocket-Extensions, or Origin. The Sec-WebSocket-Protocol header lists by preference which subprotocols the client would like to use within the WebSockets application, such as chat. These subprotocols are registered with IANA. The Sec-WebSocket-Extensions header allows for extending the basic protocol with additional protocol features. These, too, are required to be registered with IANA. Currently, there is only one registered extension: "Per-Message Compressed" WebSockets Framing Header Bit (permessage-deflate). This extension is used when adding compression functionality.
The Origin header is an optional header field in the WebSockets handshake that indicates the origin of the script making the connection. It is unique because browsers are required to include the origin header but non-browser clients only include it when it makes sense in their application. Non-browser clients can also set it to any value.

The server portion of a successful handshake is an HTTP response with “HTTP/1.1 101 Switching Protocols” as the status line. It also includes the Connection and Upgrade header fields like the client. The server responds to the client’s Sec-WebSocket-Key header with its Sec-WebSocket-Accept field. The value of this field is derived from the client’s Sec-WebSocket-Key using a set algorithm. The client uses the same algorithm to verify the value from the server. This verification proves it is a response to the client’s request and the connection is established.

The same optional header fields can be included but have slightly different meaning. The Sec-WebSocket-Protocol field contains the subprotocol the server has selected from the list the client sent. The Sec-WebSocket-Extension header value in the server response, if present, must have been present in the client’s request or the connection must fail.

Because these are still HTTP messages, they may also include standard HTTP header fields, such as cookies or authentication/authorization fields. The headers may appear in any order and if the HTTP RFC allows, may contain more than one value, or appear more than once.

Once the handshake is complete, the WebSockets packets begin within the same TCP connection. The concept of the socket allows further traffic to be initiated by either end. The protocol does not change based on the direction. Each packet is formatted the same. Figure 1 shows the wire format per the RFC (Fette & Melnikov, 2011).

Author Name, email@addressm
Figure 1 - WebSockets Wire Format

The first two bytes of a WebSockets packet are mostly flags and codes. The first one-bit flag indicates if the packet is the final fragment of a message (1) or not (0). The next three bits are reserved fields and are generally not used unless a subprotocol sets them. The next four bits are the opcode. The opcode defines how the payload data should be interpreted. The most common are text (1) and binary (2) but could also be a connection close (8), ping (9), pong (10), continuation (0), or reserved. The mask flag is a single bit that indicates if the data is masked (1) or not (0).

The last seven bits of the first two bytes are used when the payload is less than 126 bytes. The value of the seven bits is the length of the payload. If the value of the seven bits is 126, the following two bytes represent the payload length (16-bit unsigned integer). If the value of the seven bits is 127, the following eight bytes represent the payload length (64-bit unsigned integer).

If the mask flag from the first two bytes is set to one, the four-byte masking key will start at wherever the final payload length fields end. The next byte will be the start of the payload data which completes the packet.

Author Name, email@addressm
2.1.4. Spicy and Bro Scripts

The Spicy script developed for this research (ws_handshake.spicy script provided in the Appendix) parses HTTP GET request messages looking for the GET string at the start of the packet. Within the headers, the script looks for the required “Sec-WebSocket-Version” header. If both are found, it continues to parse packets within the TCP connection from the same originating IP address and port. Follow-on packets are considered WebSockets packets and parsed accordingly.

The Spicy script also parses HTTP response packets starting with “HTTP/1.1 101 Switching Proto” and looks for the required “Sec-WebSocket-Accept” header. If both are found, it continues to parse packets within the TCP connection from the same responding IP address and port. Follow-on packets are considered WebSockets packets and parsed accordingly.

The Spicy script parses WebSockets protocol packets starting with the first two bytes as a bitfield of 16 bits. A bitfield allows the parser to identify individual bits and groups of bits used by the protocol to form header fields. The script includes logic to evaluate the last eight bits of the first two bytes. The first bit of these eight bits indicates if the data is masked and a masking key is included in the packet header. The value of the last seven bits is the payload length field. The payload length field determines where the next header field begins and if it is an extended payload length field or not. It also designates where the data ends.

The Spicy parser did not function as expected when reaching the end of the data, which was also the end of the last packet. It seemed to need an additional marker called a lookahead token to know when to stop creating the list of WebSockets messages, even though reaching the end of the data should have negated the need for a lookahead token. Adding a distinguishing string to the end of the data before parsing the list of WebSockets messages created the lookahead token necessary for the parser to function correctly.

Although the script detects if the data is masked or not, it was not possible to unmask the data within Spicy. The required functionality to loop through bytes of data to apply the XOR function against the masking key is not present in Spicy. Primitive
iterator functionality does exist to iterate through bytes of data, but only if the number of iterations can be hardcoded. Spicy does offer similar functionality, such as Base64 encoding/decoding and sha256 hashing, through functions provided by its runtime library. These are written in C programming language. A knowledgeable C programmer could add the XOR function here to receive the entire data field and mask key from the Spicy script and use the looping available in C (R. Sommers, electronic mailing list message, August 2, 2017).

Once all the pertinent pieces of the WebSockets communication were parsed and made available, the Bro protocol analyzer definition script (ws_handshake.evt script provided in the Appendix) was written. The *.evt script defines significant events within the WebSockets communication. The events are triggered when a particular unit gets parsed in the Spicy script and events can pass along information about the connection. The *.evt script written for this research defines events for when a WebSockets handshake is requested, when a WebSockets handshake receives a successful response, and when WebSockets messages are detected. The message events provide different information when the WebSockets data is masked, when the data is not masked, and when there is no data.

The first Bro script (ws_handshake.bro script provided in the Appendix) was written to replicate the functionality provided by the Bro alone script (httpWSupgrade.bro). The ws_handshake.bro script takes the WebSockets connection parsed by the Spicy script and provides the same basic connection and handshake information as httpWSupgrade.bro.

The next Bro script written (ws_message.bro script provided in the Appendix) focuses on exposing the WebSockets protocol fields and payload data. The messages are separated into three different types: masked (from the client), unmasked (from the server), and no data (control messages from either host) for processing within the script, but all three log to the same file. The log file contains connection information (the unique identifier, the client IP address, the server IP address, and the server port). The entire connection record is passed to the events used in this script so that any connection field

Author Name, email@addressm
could be logged. The file also contains WebSockets information (OpCode, mask key, and payload data).

Since it was not possible to unmask the client data within Spicy, it needed to be done in Bro. Unfortunately, although Bro does have the looping capability that Spicy does not have, Bro does not have the XOR functionality. Github user JustBeck wrote an elegant solution to this same problem with his bintools.bro module (2013). His module is simply a complete lookup table for each hex byte and its corresponding XOR byte. His module was used for this research so that the masked data could be unmasked and evaluated further. As it is, the ws_message.bro Bro script and resulting log file would be inefficient and unnecessary in production but was extremely useful for testing the parser and visualizing the data.

The final Bro script (ws_messagenotnormal.bro script provided in the Appendix) demonstrates logging unexpected server responses in WebSockets data. With knowledge of the application being monitored, a regular expression was created that matches only expected responses. After a WebSockets message event is triggered, this Bro script uses the URI field from the connection and its corresponding regular expression to determine if the actual response is expected. If not, it creates a log entry. This technique was used with the DVWS Command Execution, Reflected XSS, and Error Based SQL Injection pages.

The ws_messagenotnormal.bro script also evaluates the request from the client for malicious content. It reviews input to the Error Based SQL Injection and Blind SQL injection pages attempting to identify SQL injection. It scores the message based on the number of SQL keywords it matches in the data. If the maximum threshold is surpassed, a log entry is created. The same technique is used with the Stored XSS page creating the score based on HTML keywords found in the input.
2.2. Testing

2.2.1. Setting up the lab

The testing required a Bro network monitoring system, a vulnerable WebSockets application server, and an attack system. For ease of setup and to save time, ready-made Docker images and pre-configured Linux distributions were used. The lab consisted of an Apple laptop running VMware Fusion virtualization software. An Ubuntu 17 server virtual machine was run within VMware Fusion, and Docker was installed on the Ubuntu server. A second virtual machine was created running Kali Linux 2016 Penetration Testing Distribution.

The rsmmr/hilti Docker image was simple to use as the monitoring system. It had the latest versions of Bro and Spicy, as well as all the test scripts for Spicy development. There were also protocol parsing scripts installed that could be used for reference, such as tls.spicy, bacnet.spicy, and dns.spicy. Using the rsmmr/hilti image did require some fundamental Docker knowledge and a system capable of running Docker. For this research, a standard Ubuntu 17 server virtual machine was created, and Docker was installed. The Docker.com website and online tutorials provided more than enough to use the image.

The Open Web Application Security Project (OWASP) has a sub-project called Damn Vulnerable Web Sockets (DVWS). This project provides the PHP source code for a deliberately vulnerable web application that uses WebSockets. The application is meant to be exploited. Working through the exploitation of DVWS develops valuable experience for those learning how to attack, as well as for those learning how to code more securely. Although OWASP does not provide more than the source code, there is a functioning image called tssoffsec/dvws (2017) on the public Docker hub site that was adequate for this research.

The DVWS Docker image can share the network devices of the host virtual machine which allowed for packet capture using tcpdump. Tcpdump is included with the Ubuntu server install but is not a part of the DVWS Docker image. Once running, the image ran through the database setup steps automatically. This process could also be invoked by visiting the application’s setup.php page, allowing for quick resetting of the database.

Author Name, email@addressm
database after testing. The only other setup required was to configure the attacking machine for name resolution of the DVWS.local hostname. Setting it in the /etc/hosts file on the attacking machine was the easiest way to do this.

The Kali virtual machine was upgraded to the latest SQLmap version, and the websocket-client third party package was installed. This package allowed SQLmap to interact with the DVWS for Blind SQL attacks.

The Apple laptop was also an attack machine. Firefox web browser and Burp Suite free version were installed and used to launch the attacks. WebSockets data was viewed with Firefox’s Developer Tools and Burp Suite’s proxy.

2.2.2. Determining the tests

Once the environment was configured, tests were selected to address relevant security risks related to WebSockets applications. The OWASP list of Top 10 Most Critical Web Application Security Risks and the DVWS project were referenced to determine which risks were relevant. The Top 10 list is created by gathering vulnerabilities from hundreds of organizations and over 50,000 real-world applications. OWASP estimates the exploitability, detectability, and impact of each vulnerability when determining overall criticality. The vulnerabilities are then prioritized by criticality, and the top ten are detailed in the report (OWASP, 2017).

The Top 10 list references WebSockets within the topic of “Underprotected APIs” because WebSockets allow web applications to create their own protocol and data formats which can introduce vulnerabilities similar to those introduced by APIs. Testing for vulnerabilities in this overarching category should be the same as testing the full web application. It should include all the same tests described in the other Top 10 categories (OWASP, 2017). The DVWS project demonstrates how the vulnerabilities from the Top 10 list can apply to WebSockets applications. The DVWS project hosts eight vulnerable PHP scripted pages that use WebSockets. These pages are susceptible to Brute Force, Command Execution, Cross-Site Request Forgery (CSRF), File Inclusion, Error SQL Injection, Blind SQL Injection, Reflected Cross-Site Scripting (XSS), and Stored XSS.

Author Name, email@addressm
From the vulnerable pages available for testing in the DVWS project, all but the Brute Force page were selected to use for testing the Spicy scripts. The Brute Force test required additional server resources that were not available during the research, and the tests would not have provided additional information that significantly impacted the research findings.

### 2.2.3. Testing

The Testing WebSockets portion of the new OWASP Testing Guide v4 provided a starting point for conducting the tests. It discussed the categories of vulnerabilities to test, such as Authentication and Input Sanitization, and their corresponding issues on the Top 10 list. However, it did not provide very detailed steps for conducting the tests, or instructions for the tools it recommended for use in testing. For that information, the “Damn Vulnerable Web Sockets Walkthrough” blog article was indispensable (Dardas, 2017).

Dardas’s DVWS walkthrough article detailed the steps he used to exploit each of the vulnerable DVWS scripts, and provided links to the exploit scripts he wrote. Dardas’s instructions and scripts were used in this research to simplify and standardize the testing process. Specifically, this study used his CSRF, File Inclusion, Error SQL Injection, Blind SQL Injection, Command Execution, and Reflected XSS exploits and instructions.

For repeatability, each attack was captured in a packet capture file using tcpdump on the virtual machine running Docker. Bro then evaluated the packet captures using the .spicy, .evt, and .bro scripts. The process was iterative where the scripts were evaluated and modified until the desired outcome was reached.

The Apple attack machine used Burp Suite as a proxy to capture and modify WebSockets traffic going to the victim PHP pages for manual exploits. Automated exploits written by Dardas were run in the attack machine’s Firefox browser. These were scripts written as JavaScript in an HTML file that only needed to be opened in the browser to launch the attack. SQLMap on the Kali virtual machine was used against the Blind SQL Injection page.

Author Name, email@addressm
Script performance was also evaluated. All the testing packet captures were combined and then multiplied using a Scapy script so that each attack stream would be seen by Bro 1,000 times. The script modified the packet times so that Bro would not see the additional traffic as duplicate packets. This combined pcap file was then run through Bro using the –Q argument to output execution time and memory usage summaries. The entire Bro command was prefaced with the GNU time command to also output system usage statistics. The combined pcap file was run three times using the Bro native HTTP parser and httpWSupgrade.bro script, and three times using the Spicy WebSockets parser and ws_handshake.bro script. The results were averaged and then compared.

2.3. Script and Log Comparison

The WebSockets data that the existing HTTP parser can provide is limited to the handshake headers. While this can provide some useful information for logging WebSockets connections, it provides very little in detecting attacks that use WebSockets. Many attacks take advantage of vulnerabilities in the application data carried within the WebSockets payload. The existing HTTP parser does not parse the WebSockets messages after the initial handshake since they are no longer using HTTP.

Parsing the traffic with a Spicy generated WebSockets protocol analyzer also provides the same handshake header information that the HTTP protocol analyzer provides. However, this research shows that Bro scripts can build on the WebSockets events created by the Spicy analyzer to provide WebSockets distinct data. This visibility into the WebSockets data is the first step in improving attack detection.

2.4. Detection and Effectiveness Comparison

Bro alone can read the WebSockets handshake traffic based on events created by the HTTP protocol parser. Bro with the Spicy parser can also read WebSockets handshake traffic with events generated by the Spicy script. Since both techniques produce an output file with the same WebSockets handshake data, their effectiveness is the same. However, there are currently no WebSockets attacks that can be detected solely by monitoring the handshake traffic.

Author Name, email@addressm
Before being able to detect malicious WebSockets traffic, the data in the traffic must be readable. Without a WebSockets parser, Bro cannot read the data. In this sense, comparing the effectiveness of WebSockets attack detection between Bro alone and Bro with the Spicy parser becomes quite simple. Bro with the Spicy parser can detect attacks that Bro alone cannot.

Although there are clear advantages to using Bro with the Spicy generated WebSockets parser to detect malicious traffic, it does not come without drawbacks. When evaluated for performance, the additional parsing added considerably to the overall processing time and memory requirements (Results provided in the Appendix). The Spicy scenario required about two and a half times the processing time (user plus system times) and one and a half times the memory.

The Spicy parser generation that compiles at runtime added to the time and memory required for Bro initialization. Bro with Spicy took about two and a half times longer to initialize and used about one and a half times the amount of memory used by Bro alone.

The Spicy scripts in this study were written without consideration for their impact on performance. It is expected that performance would be improved with a more efficient Spicy script, but it is unlikely that the difference could be eliminated. The additional work required to parse the data and not just the Websocket handshake packets will inevitably have some impact. Further investigation into the efficiency of Spicy generated parsers compared to those written in C or with BinPAC would be required to achieve the best performance.

Judicious use of the Spicy WebSockets parser could still add a benefit that overcomes the performance impact. For example, it would be suitable for evaluating traffic offline or during network forensic investigations when performance is not as crucial. It could also be strategically placed on the network to limit the traffic it monitors, such as in front of a critical WebSockets application.
3. Conclusion

WebSocket usage is growing, and so are the number of WebSocket exploits. Therefore, security analysts need the capability to evaluate network traffic for WebSocket attacks. Spicy provides a relatively easy and accessible solution for bridging the gap between what can be seen with Bro alone and what is currently on the wire hiding in new protocols such as WebSockets. Spicy generated parsers allow for visibility into WebSocket traffic that Bro alone does not have. This visibility enables detection of WebSocket attacks that would otherwise not be detected. However, Spicy does introduce significant performance impacts to both computing resources and processing time. Spicy is also not currently supported for production usage. This research shows that Spicy is a promising solution to network monitoring and attack detection in new protocols. Once Spicy is released for production use, it will likely require strategic employment to counter performance impacts, but will still provide benefits to security professionals.
A Spicy Approach to WebSockets: Enhancing Bro’s WebSockets Network Analysis by Generating a Custom Protocol Parser with Spicy

References


Author Name, email@addressm
A Spicy Approach to WebSockets: Enhancing Bro’s WebSockets Network Analysis
by Generating a Custom Protocol Parser with Spicy


A Spicy Approach to WebSockets: Enhancing Bro’s WebSockets Network Analysis by Generating a Custom Protocol Parser with Spicy


Author Name, email@addresm
A Spicy Approach to WebSockets: Enhancing Bro’s WebSockets Network Analysis by Generating a Custom Protocol Parser with Spicy


Author Name, email@addressm

© 2017 The SANS Institute
Author retains full rights.
Appendix

1. httpWSupgrade.bro script
2. ws_handshake.spicy script
3. ws_handshake.evt script
4. ws_handshake.bro script
5. ws_message.bro script
6. ws_messagenotnormal.bro script
7. Performance Comparison Results
Appendix 1. httpWSupgrade.bro Script

```bro
# This script upgrades HTTP connections to WebSockets
# Author: Jennifer Gates
# Date: August 2017

# Using the http_all_headers event, creates the httpWSupgrade.log file. The
# log file provides WebSockets handshake header information such as the host,
# URL, origin, location, subprotocols, and extensions, as well as basic
# connection information such as the timestamp, UID, server IP address, server
# port, and client IP address.
# The script loads the base/protocols/http and base/protocols/conn
# modules into the namespace.
# This function creates a new stream.

def custom_protocol() {
    # Define the record type that will contain the data to log.
    enum Log::ID := { LOG, };
    type Info := record {
        ws_handshake: string &log;
        ws_timestamp: time &log;
        ws_client: string &log;
        ws_server: string &log;
        ws_host: string &log;
        ws_url: string &log;
        ws_useragent: string &log;
        ws_acceptkey: string &log;
        ws_origin: string &log;
        ws_location: string &log;
        ws_protocol: string &log;
        ws_extensions: string &log;
    };

    event bro_init(priority=5) {
        # This creates a new stream.
        Log::create_stream(httpWSupgrade::LOG, {columns=Info, path="httpWSupgrade"});
    }

    # Add a new field to the connection record so that data is accessible in
    # the variety of event handlers.
    record connection := { info(Info) &optional; };

    # Use http_all_headers event as defined in Bro_HTTP.events.bif.bro. It returns
    # a list of headers indexed by order in packet and containing name/value pairs
    event http_all_headers(c: connection, is_orig: bool, hlist: mime_header_list) {
        # Initialize non-required fields or fields not always present in a packet
        local origin = "" -;
        local location = "" -;
        local acceptkey = "" -;
        local useragent = "" -;
        local handshake = "" -;
    }
}
```

Author Name, email@email@address
A Spicy Approach to WebSockets: Enhancing Bro’s WebSockets Network Analysis
by Generating a Custom Protocol Parser with Spicy

local wsproto = '';
local wsexts = '';
local srvip = addr;
local citip = addr;
local srvpt: port;

# look through all headers for handshake headers to log
for [i in hlist]
{
    # start with a blank handshake until logic determines if this header
    # is a websocket handshake header
    handshake = '';

    # if this is a client request to handshake, the client must indicate
    # the version as 13 per the RFC
    if ["SEC-WEBSOCKET-VERSION" in hlist[i]$name & & "13" in hlist[i]$value ]
        
        handshake = "REQUEST";

    for [y in hlist]
    {
        if ["SEC-WEBSOCKET-KEY" in hlist[y]$name ]
            
            acceptkey = hlist[y]$value;

        if ["ORIGIN" in hlist[y]$name ]
            
            origin = hlist[y]$value;

        if ["USER-AGENT" in hlist[y]$name ]
            
            useragent = hlist[y]$value;

        # In the Request, there could be multiple protocols headers
        if ["SEC-WEBSOCKET-PROTOCOL" in hlist[y]$name ]
            
            if ( wsproto == "-" )
                
                wsproto = hlist[y]$value;
            else
                
                wsproto = hlist[y]$value;

        # In the Request, there could be multiple extensions headers
        if ["SEC-WEBSOCKET-EXTENSIONS" in hlist[y]$name ]
            
            if ( wsexts == "-" )
                
                wsexts = hlist[y]$value;
            else
                
                wsexts = hlist[y]$value;

    }

    # if this is a server response to a successful handshake, it will have
    # a status code of 101 to test if a field that is optional has
    # assigned a value, use the ? operator
    if ["UPGRADE" in hlist[i]$name & & "websocket" in to_lower(hlist[i]$value) & & cshttp?status_code & & cshttp?status_code == 101]
        
        handshake = "REPLY";

    for [x in hlist]
    {
        if ["SEC-WEBSOCKET-ACCEPT" in hlist[x]$name ]
            
            acceptkey = hlist[x]$value;

        if ["WEBSOCKET-ORIGIN" in hlist[x]$name ]
            
            origin = hlist[x]$value;

    }

Author Name, email@addressm
A Spicy Approach to WebSockets: Enhancing Bro’s WebSockets Network Analysis by Generating a Custom Protocol Parser with Spicy

Author Name, email@address
A Spicy Approach to WebSockets: Enhancing Bro’s WebSockets Network Analysis by Generating a Custom Protocol Parser with Spicy

Appendix 2. ws_handshake.spicy Script

```spicy
module WS_HANDSHAKE;

import Spicy;
import "MULTI-C" void Multi::terminate();

const BeforeColon = /\[\-]+\/;
const Colon = /\;/;
const DataValue = /\^\[\x0d\x20\x8a]+\;/;
const LineEnd = /\x0d\x20/;
const DataEnd = b'\x53\x08\x59\x6a\x6c\x67';
global gtotaldata: bytes = b'';

# function to add a distinguishing byte at the end of the ws packets
# during parsing to allow list to properly terminate
bytes terminate(b: bytes) {
    return b + DataEnd;
}

# primary unit to store client originator Websocket handshake traffic
# uses sink to parse follow on websocket protocol data packets
export type WS_Handshake = unit {
    get : \("GET\"\)["GET"] /;
dvalue : DataValue;
headers : list-Header=;
end_of_hdrs : /\x0d\x20\x8a/;

ws_data : bytes Good &convert=terminate(ss) &Transient -> self.hs_sub;
on %init {
    self.hs_sub.connect(new HS_Sub(self));
}

var hs_sub: sink;
var messages: list-WS_Message=;
var totaldata: bytes;

# steps parsing if HTTP GET request is not initiating Websocket handshake
# joins all headers into a string and searches for Sec-WebSocket-Version header
on headers {
    local joinedheaders: bytes;
    joinedheaders = b'', join(self.headers);
    joinedheaders = joinedheaders.lower();
    if (\| joinedheaders.match(\"^sec-websocket-version.\") \| == 0){
        Multi::terminate();
    }
}

# combines data fields from all websocket packets into one string
on %done {
    self.totaldata = gtotaldata;
    gtotaldata += b'';
    self.ws_sub.close();
}
};

# sub unit to put originator websocket protocol packets into a list
export type HS_Sub = unit{handshake: WS_Handshake} {
    ws_msgs : list<WS_Message> {
        handshake.messages = self.ws_msgs;
    }
    : DataEnd;
};

# primary unit to store server/responder Websocket handshake traffic
# uses sink to parse follow on websocket protocol data packets
```

Author Name, email@address
A Spicy Approach to WebSockets: Enhancing Bro’s WebSockets Network Analysis
by Generating a Custom Protocol Parser with Spicy

export type WS_Handshake_Success = unit {
  success : /"HTTP/1.1 101 Switching Protocols";
  dvalue : DataValue;
  srvheaders : List<Header>;
  end_of_hdrs : /[\x0d\x0a\x0d\x0a]/;
  wss_data : bytes & convert terminate(S5) & transient -> self.hss_sub;

  on %unit {
    self.hss_sub.connect(new SSS_Sub(self));
  }
}

var hss_sub: sink;
var messages: List<WS_Message>;
var totaldata: bytes;

# stops parsing if HTTP reply is not responding to Websocket handshake
# joins all headers into a string and searches for Sec-WebSocket-Accept header
on srvheaders {
  local sjoinedheaders: bytes;
  sjoinedheaders = "\n".join(self.srvheaders);
  sjoinedheaders = sjoinedheaders.lower();
  if (sjoinedheaders.match(/\+sec-websocket-accept\./) === 0) {
    Httpt::terminate();
  }
}

# combines data fields from all websocket packets into one string
on done {
  self.totaldata = totaldata;
  totaldata = b" ";
  self.hss_sub.close();
}

# sub unit to put responder websocket protocol packets into a list
export type HSS_Sub = unit (handshake: WS_Handshake_Success) {
  wss_msg: List<WS_Message> =
    handshake.smessages = self.wss_msg;
}

# unit to parse http headers in handshake packets
export type Header = unit {
  name : String;
  value : DataValue;
}

# sub unit to parse WS protocol fields from websockets packet list
export type WS_Message = unit {
  # start with first 2 bytes as a bitfield to get access to bit level flags
  first2B: bitfield(16) {
    fin: 0;
    rv1: 3;
    rv2: 2;
    rv3: 3;
    op: 4..7;
    mask: 8;
    payl1: 9..15;
  } sbitorder = Spicy::BitOrder::MSB0;

  # determine payload length and start point for next field
  switch (self.first2B.payl1) {
    126 -> extpay: ExtPey;
    127 -> extpay: ExtPeyC;
    + -- : void;
  };

  # parse out masking key if payload is masked (from client)
  switch (self.first2B.mask) {
    1 -> maskkey: bytes & length=4;
    0 -> : void;
  };

  # grab the payload data, leave unmasking to bro script
  switch (self.first2B.payl1) {
    126 -> data: bytes & length=self.extpay.pay2 ;
  }

Author Name, email@address
A Spicy Approach to WebSockets: Enhancing Bro’s WebSockets Network Analysis
by Generating a Custom Protocol Parser with Spicy

Author Name, email@addressm
Appendix 3. ws_handshake.evt Script

```plaintext
grammar ws_handshake.spicy;

protocol analyzer WS_HANDSHAKE over TCP:
    parse originator with WS_HANDSHAKE::WS_Handshake,
    parse responder with WS_HANDSHAKE::WS_Handshake_Success,
    port 80/tcp,
    port 8008/tcp,
    port 12345/tcp,
    port 59966/tcp,
    port 59964/tcp,
    port 44394/tcp;

on WS_HANDSHAKE::WS_Handshake-->
    event ws_handshake($conn, self.headers);

on WS_HANDSHAKE::Header-->
    event header($conn, self.name, self.value);

on WS_HANDSHAKE::WS_Handshake-->
    event all.headers($conn, self.headers, self.value);

on WS_HANDSHAKE::WS_Handshake_Success-->
    event all.headers($conn, self.svrheaders, self.value);

on WS_HANDSHAKE::WS_Message::first28 -->
    event ws_messages($conn, self.first28.op, self.first28.mask);

on WS_HANDSHAKE::WS_Message if {{ self?.maskkey } & (self?.data)} -->
    event ws_maskedmessage($conn, self.first28, self.maskkey, self.data);

on WS_HANDSHAKE::WS_Message if {{ !self?.maskkey } & (self?.data)} -->
    event ws_unmaskedmessage($conn, self.first28, self.data);

on WS_HANDSHAKE::WS_Message if {{ !self?.maskkey } & (!self?.data)} -->
    event ws_nodatamessage($conn, self.first28);

# WebSockets Protocol Analyzer Definition $script
#
# Jenniffer Gates
# August 2017
#
# The Bro protocol analyzer definition script (*.evt) provides the interface
# between Bro and the Spicy script. It specifies the grammar file that contains
# the Spicy script and where to hook into Bro’s traffic processing. It then
# defines the Bro events to create based on the parsed data.
#
# event triggers when the WS_Handshake type is parsed. The connection and
# the list of headers are passed to it.
# on WS_HANDSHAKE::WS_Handshake-->
#    event ws_handshake($conn, self.headers);
# event triggers when the Header type is parsed. The connection,
# the header name, and its value are passed to it.
# on WS_HANDSHAKE::Header-->
#    event header($conn, self.name, self.value);
# event triggers when the WS_Handshake type is parsed. The connection,
# the list of headers, and the http request line are passed to it.
# on WS_HANDSHAKE::WS_Handshake-->
#    event all.headers($conn, self.headers, self.value);
# event triggers when the WS_Handshake_Success type is parsed. The connection,
# the list of headers, and the http response line are passed to it.
# on WS_HANDSHAKE::WS_Handshake_Success-->
#    event all.headers($conn, self.svrheaders, self.value);
# event triggers when the first28 WS_Message type is parsed. The connection,
# the opcode, and the mask flag are passed to it.
# on WS_HANDSHAKE::WS_Message::first28 -->
#    event ws_messages($conn, self.first28.op, self.first28.mask);
# event triggers when the WS_Message type is parsed and both the mask key and data
# fields are populated. The connection, the mask key, and the data are passed to it.
# on WS_HANDSHAKE::WS_Message if {{ self?.maskkey } & (self?.data)} -->
#    event ws_maskedmessage($conn, self.first28, self.maskkey, self.data);
# event triggers when the WS_Message type is parsed, mask key is not set, and data
# fields is populated. The connection, the first28, and the data are passed to it.
```

Author Name, email@addressm
A Spicy Approach to WebSockets: Enhancing Bro’s WebSockets Network Analysis by Generating a Custom Protocol Parser with Spicy

```javascript
81  // on WS_HANDSHAKE::WS_Message if (!self?.maskkey) & & (self?.data) ->
82  // event ws_unmaskedmessage($conn, self.first2B, self.data);
83  // event triggers when the WS_Message type is parsed, mask key is not set, and data
84  // fields is not populated. The connection and the data are passed to it.
85  // on WS_HANDSHAKE::WS_Message if (!self?.maskkey) & & (self?.data) ->
86  // event ws_modatamessage($conn, self.first2B);
```

Author Name, email@addressm
Appendix 4. ws_handshake.bro Script

```bro
# A spicy approach to WebSockets: Enhancing Bro's WebSockets Network Analysis by Generating a Custom Protocol Parser with Spicy

# WebSockets Handshake Bro script
# Jennifer Gates
# August 2017
#
# This script takes the WebSockets connection parsed by the Spicy script and, using the allheaders event, creates the ws_handshake.log file. The log file provides WebSockets handshake header information such as the host, URI, origin, location, subprotocols, and extensions, as well as basic connection information such as the timestamp, UID, server IP address, server port, and client IP address.
#
# load processes the _load_.bro scripts in the directories loaded which basically includes 
# @load base/protocols/http
# @load base/protocols/conn
#
# create namespace
# module WS_HANDSHAKE;

export {

    # Create an ID for our new stream.
    redefine enum Log::ID = { LOG };

    # Define the record type that will contain the data to log.
    type Info: record {
        # Indicates if log info is for WebSocket Handshake Request or Reply
        ws_handshake: string &log;
        # Timestamp for when the request happened
        ws_ts: time &log;
        # Unique ID for the connection
        ws_uid: string &log;
        # Client IP requesting WebSockets
        ws_client: addr &log;
        # Server IP providing WebSockets
        ws_srvr: addr &log;
        # Server port providing WebSockets
        ws_svrport: port &log;
        # Value of the HOST header
        ws_host: string &log;
        # Value of the User-Agent header from the client
        ws_useragent: string &log;
        # Value of the client's SEC-WEBSOCKET-KEY if a request, still base64 encoded
        ws_acceptkey: string &log;
        # Value of the ORIGIN header
        ws_origin: string &log;
        # Value of the LOCATION header
        ws_location: string &log;
        # Value of the SEC-WEBSOCKET-PROTOCOL header
        ws_protocol: string &log;
        # Value of Sec-WebSocket-Extensions if present
        ws_extensions: string &log;
    };

    event bro_init() @priority=5 {
        # Create the stream. this adds a default filter automatically
        Log::create_stream(WS_HANDSHAKE::LOG, $columns=Info, $path="ws_Handshake");
    }

    # add a new field to the connection record so that data is accessible in variety of event handlers
    redefine record connection += {
        ws_handshake: Info &optional;
    };

    # define for Bro the record that will be passed in from spicy parser in the headers list
    type BroHdr: record {
        name: string;
        value: string;
    };

    # define for Bro the vector that will be passed in from spicy parser as the headers list
    type BroHdrs: vector of BroHdr;
```

Author Name, email@addressm
# define the first2B tuple for Bro for the record that will be passed in from spicy parser

type Brofirst2B: record {
    fin: count;
    rsv1: count;
    rsv2: count;
    rsv3: count;
    op: int;
    mask: count;
    payl: int;
};

# define for Bro the record that will be passed in from spicy parser in the ws messages list
# <first2B={fin=1, rsv1=0, rsv2=0, rsv3=0, op=1, mask=1, payl=12}>

type BroMsg: record {
    first2B: Brofirst2B;
    pay2: int &optional;
    pay3: int &optional;
    maskkey: string &optional;
    data: string &optional;
};

# define for Bro the vector that will be passed in from spicy parser as the ws messages list

type BroMsgs: vector of BroMsg;

# event that is basically same data as http parser alone with custom bro script, which this
# code is from

event allheaders(c: connection, hlist: BroMsgs, reqlineData: string)
{
    # initialize non-required fields or fields not always present in a packet
    local uri = "- ";
    local host = "- ";
    local origin = "- ";
    local location = "- ";
    local acceptkey = "- ";
    local useragent = "- ";
    local handshakes = "- ";
    local wsprotocol = "- ";
    local wevents = "- ";
    local svrip = addr;
    local cltip = addr;
    local svrp: port;

    # look through all headers for handshake headers to log
    for (i in hlist)
    {
        # start with a blank handshake until logic determines if this header is a websocket
        # handshake header
        handshakes = "- ";

        # if this is a client request to handshake, the client must indicate the version as
        # 13 per the RFC
        if ("sec-websocket-version" in to_lower(hlist[i]$name) & 66 "13" in hlist[i]$value )
        {
            handshakes = "REQUEST";
        }

        local reqlineDataSplit = split_string_all(reqlineData, /HTTP/);
        uri = reqlineDataSplit[0];

        for (y in hlist)
        {
            if ("sec-websocket-key" in to_lower(hlist[y]$name) )
            {
                acceptkey = hlist[y]$value;
            }

            if ("host" in to_lower(hlist[y]$name) )
            {
                host = hlist[y]$value;
            }

            if ("origin" in to_lower(hlist[y]$name) )
            {
                origin = hlist[y]$value;
            }

            if ("user-agent" in to_lower(hlist[y]$name) )
            {
A Spicy Approach to WebSockets: Enhancing Bro’s WebSockets Network Analysis by Generating a Custom Protocol Parser with Spicy

```c
...
...

// if a handshake header was found, log the handshake header information to ws_handshake.log
if ([handshake] > 1) {
  {
    Log format local rec: WS_HANDSHAKE::Info = [ws_t=websocket, ws_uid=uid, ws_client=id, ws_protocol=protocol, ws_extension=extension, ws_handshake=handshake]
    for (ws_handshake = rec;
      Log::write(WS_HANDSHAKE::LOG, rec);
    }

Author Name, email@addressm
```
Appendix 5. ws_message.bro Script

```bro
# A spicy approach to WebSockets: Enhancing Bro’s WebSockets Network Analysis
# by generating a custom protocol parser with Spicy

# WebSockets Message Bro script
# August 2017
#
# Script to expose the WebSockets protocol fields and payload data. The
# messages are separated into three different types: masked (from the client),
# unmasked (from the server), and no data (control messages from either host)
# for processing within the script, but all three log to the WS_Message.log file.
# The log file contains the connection ID, the client IP, the server IP, the
# server port, OpCode, mask key, and payload data.
#
# load processes the __load__.bro scripts in the directories loaded
# load base/protocols/http
# load base/protocols/conn
# load bintools
#
# create namespace
module WSMESSAGE;

export {
  # Create an ID for our new stream.
  redefine enum Log::ID => { LOG, };

  # Define the record type that will contain the data to log.
  type Info::record {
    # Unique ID for the connection
    ws_uid: string &log;
    # Client IP requesting WebSocket
    ws_client: addr &log;
    # Server IP providing WebSocket
    ws_srvr: addr &log;
    # Server port providing WebSocket
    ws_srvp: port &log;
    # Opcode indicating if text, binary, etc
    ws_OPCODE: count &log;
    # Maskkey used by client to XOR mask data
    ws_maskkey: string &log;
    # Data in websocket packet
    ws_data: string &log;
  };
}

event bro_init() { &priority=5
  # Create the stream. this adds a default filter automatically
  Log::create_stream([WSMESSAGE::LOG, $columns=Info, $path="WS_Message"]);
}

# add a new field to the connection record so that data is accessible in variety of event handlers
redefine record connection += {
  ws_message: Info OPTIONAL;
};

# define the first2B tuple for Bro for the record that will be passed in from spicy parser
type Brofirst2B: record {
  fin: count;
  rsv1: count;
  rsv2: count;
  op: count;
  mask: count;
  payload: count;
};

# log information from a message with masked data
event ws_maskedmessage(c: connection, first2B: Brofirst2B, maskkey: string, data: string) {
  local nkey = "-";
  local wdata = "-";
  local xdata = "";
  if [maskkey] > 1 {
    nkey = maskkey;
  }
  local ct: count = 0;
```

Author Name, email@addressm
A Spicy Approach to WebSockets: Enhancing Bro’s WebSockets Network Analysis
by Generating a Custom Protocol Parser with Spicy

# XOR lookup function provided by https://github.com/justbeck/bro-xorpe/blob/master/binTools.bro
for ( byte anData ) {
    xordata = BinTools::xorByte( byte, mkey[(ct % 4)]);
    ct = ct - 1;
}
wsdata = xordata;

# Log format
local rec = WS_MESSAGE::Info = [sws_uid=c$uid, sws_client=c$1d$orig_h, sws_srv=c$1d$resp_h,
sws_srvp=c$1d$resp_p, sws_opcode=first2B$op, sws_maskkey=mkey, sws_data=wsdata];
c$ws_message = rec;
Log::write(WS_MESSAGE::LOG, rec);

# log information from a message with unmasked data
event ws_unmaskedmessage(c: connection, first2B: Brofirst2B, data: string) {
    local mkey = " - ";
    local wsdata = data;
    # Log format
    local urec = WS_MESSAGE::Info = [sws_uid=c$uid, sws_client=c$1d$orig_h, sws_srv=c$1d$resp_h,
sws_srvp=c$1d$resp_p, sws_opcode=first2B$op, sws_maskkey=mkey, sws_data=wsdata];
c$ws_message = urec;
    Log::write(WS_MESSAGE::LOG, urec);
}

# log information from a message with no data
event ws_nodatamessage(c: connection, first2B: Brofirst2B) {
    local wsdata = " - ";
    # Log format
    local nrec = WS_MESSAGE::Info = [sws_uid=c$uid, sws_client=c$1d$orig_h, sws_srv=c$1d$resp_h,
sws_srvp=c$1d$resp_p, sws_opcode=first2B$op, sws_maskkey=mkey, sws_data=wsdata];
c$ws_message = nrec;
    Log::write(WS_MESSAGE::LOG, nrec);
}
Appendix 6. ws_messagenotnormal.bro Script

```
# This script is for testing WebSockets messages that are not normal activity.
# It logs messages that are not expected or known to be malicious behavior.
# It uses the Bro parser to create a custom protocol parser with Spicy.

# Load base protocols/connections
@load base/protocols/conn

# Load base protocols/http
@load base/protocols/http

# Load policy/framework/intel/seen
@load policy/framework/intel/seen

# Define some constants for the ideas and regular expressions.
const CustomURI1 = "/command-execution";
const CustomURI2 = "/reflected-xss";
const CustomURI3 = "authenticate-user";
const CustomURI4 = "welcome-to-your-account";
const CustomURI5 = "authenticate-user-blind";

# Create namespace.
module WS_MESSAGE_NOTNORMAL;

export {
    # Create an ID for our new stream.
    redefine enum Log::ID { LOG, CUTOFF, FREE, WARNING, ERROR, CRITICAL, EMERGENCY);

    type Info::record {
        # Unique ID for the connection.
        Info::id: ID;
        # Client IP requesting WebSocket.
        ws::client: addr &log;
        # Server IP providing WebSocket.
        ws::srv: addr &log;
        # Opcode indicating if text, binary, etc.
        ws::opcode: uint &log;
        # Maskkey used by client to XOR mask data.
        ws::maskkey: string &log;
        # URI in websocket packet.
        ws::uri: string &log;
        # Data in websocket packet.
        ws::data: string &log;
    }

    # Append a new notice value to the Notice::Type enum.
    redefine enum Notice::Type += [ Unexpected_response ];
    redefine enum Notice::Type += [ SQL_injection_words ];
}

event bro_init() &priority=5
{
    # Create the stream. This adds a default filter automatically.
    log::create_stream("WS_MESSAGE_NOTNORMAL::LOG", {columns=Info, $path=\"WS_MessageNotNormal\"});
}

# Add new field to the connection record so that data is accessible in a variety of event handlers.
redefine record connection { ws::messagenotnormal: Info &optional; }
```
A Spicy Approach to WebSockets: Enhancing Bro’s WebSockets Network Analysis by Generating a Custom Protocol Parser with Spicy

```

```

Author Name, email@address
A Spicy Approach to WebSockets: Enhancing Bro’s WebSockets Network Analysis by Generating a Custom Protocol Parser with Spicy

```python
xordata = binascii.hexlify(byte, mkey[ct % 4]));
ct = ct + 1;
}

wsdata = xordata;

# detect SQL injection and HTML code words in the input and develop a score
local SQLscore = 0;
local HTMLscore = 0;

if ((cshhttpuri == CustomURI3) || (cshhttpuri == CustomURI5)) {
  # Log format
  local urec3: WS_MESSAGE_NORMAL::Info = { $ws_uid=auid, $ws_client=client_id, $ws_srv=server_addr, $ws_srvp=server_port, $ws_epocode=first256op, $ws_passkey=mkey, $ws_url=cshhttpuri, $ws_data=wsdata};
  cs_httpstatusnormal = urec3;

  # parse the input into username and password
  local array3 = split_string(wsdata, /, /);

  # parse out input for username and decode from base64
  local usernameencoded = split_string(array3[0], //)[1][1][1][2];
  local username3 = decode_base64(usernameencoded);
  local wordsinusername3 = split_string(username3, /);

  # test for common SQL injection words in username
  for (w in wordsinusername3) {
    if (SQLInjectionRegEx in to_lower(wordsinusername3[w])) {
      SQLscore = SQLscore + 1;
    }
  }

  # parse out input for username and decode from base64
  local pass3encoded = split_string(array3[1], //)[1][1][1][2];
  local pass3 = decode_base64(pass3encoded);
  local wordsinpass3 = split_string(pass3, /);

  # test for common SQL injection words in password
  for (w in wordsinpass3) {
    if (SQLInjectionRegEx in to_lower(wordsinpass3[w])) {
      SQLscore = SQLscore + 1;
    }
  }

  if (SQLscore > 4) {
    log_write(WS_MESSAGE_NORMAL::LOG, urec3);
    NOTICE($note=WS_MESSAGE_NORMAL::SQL Injection_words, $msg = fmt("SQL Injection words found going to 4", cshhttpuri), $sub = fmt("username = %s and pass = %s", username3, pass3), $connm); }
  }

  if (cshhttpuri == CustomURI4) {
    # Log format
    local urec4: WS_MESSAGE_NORMAL::Info = { $ws_uid=auid, $ws_client=client_id, $ws_srv=server_addr, $ws_srvp=server_port, $ws_epocode=first256op, $ws_passkey=mkey, $ws_url=cshhttpuri, $ws_data=wsdata};
    cs_httpstatusnormal = urec4;

    # parse the input into name and comments
    local array4 = split_string(wsdata, /, /);

    # parse out input for name
    local name4 = split_string(array4[0], //)[1][1][1][1];

    # test for HTML reg ex in name
    for (r in HTMLRegExs) {
      if (r in to_lower(name4)) {
        HTMLscore = HTMLscore + 1;
      }
    }

    # parse out input for comments
    local comments4 = split_string(array4[1], //)[1][1][1][1];

    # test for HTML reg ex in comments
    for (r in HTMLRegExs) {
      if (r in to_lower(comments4)) {
        HTMLscore = HTMLscore + 1;
      }
    }

    if (HTMLscore == 4) {
      log_write(WS_MESSAGE_NORMAL::LOG, urec4);
      NOTICE($note=WS_MESSAGE_NORMAL::SQL Injection_words, $msg = fmt("HTML code found going to 4", CustomURI4), $sub = fmt("name = %s and comments = %s", name4, comments4), $connm); }
  }
}
```

Author Name, email@address
Appendix 7. Performance Comparison Results

<table>
<thead>
<tr>
<th>GNU time output</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>averages</th>
</tr>
</thead>
<tbody>
<tr>
<td>User time</td>
<td>480.8</td>
<td>475.84</td>
<td>491.11</td>
<td>482.583333</td>
</tr>
<tr>
<td>(seconds)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>System time</td>
<td>187</td>
<td>195.54</td>
<td>203.7</td>
<td>195.413333</td>
</tr>
<tr>
<td>(seconds)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Percent of CPU</td>
<td>129%</td>
<td>127%</td>
<td>127%</td>
<td>131.6666667</td>
</tr>
<tr>
<td>this job got</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Elapsed (wall</td>
<td>9:06:63</td>
<td>8:45:07</td>
<td>8:35:49</td>
<td></td>
</tr>
<tr>
<td>clock) time m:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ss</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum resident</td>
<td>217336</td>
<td>217284</td>
<td>217540</td>
<td>217386.6667</td>
</tr>
<tr>
<td>set size (kbytes)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Minor (reclaiming</td>
<td>58861</td>
<td>60883</td>
<td>59774</td>
<td>59839.3333</td>
</tr>
<tr>
<td>a frame) page</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>faults</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Voluntary context</td>
<td>43277768</td>
<td>46541947</td>
<td>47986737</td>
<td>45935484</td>
</tr>
<tr>
<td>switches</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Involuntary</td>
<td>12537173</td>
<td>16116258</td>
<td>17470239</td>
<td>15374556.67</td>
</tr>
<tr>
<td>context switches</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>File system</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>inputs</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>File system</td>
<td>1121320</td>
<td>1121376</td>
<td>1121328</td>
<td>1121341.333</td>
</tr>
<tr>
<td>outputs</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Page size</td>
<td>4096</td>
<td>4096</td>
<td>4096</td>
<td>4096</td>
</tr>
<tr>
<td>(bytes)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bro -Q</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>averages</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialization</td>
<td>3.572282</td>
<td>2.842934</td>
<td>3.23646</td>
<td>3.217225333</td>
</tr>
<tr>
<td>initialization</td>
<td>212M/63M</td>
<td>212M/63M</td>
<td>212M/63M</td>
<td>212M/63M</td>
</tr>
<tr>
<td>total time</td>
<td>546.594225</td>
<td>524.668284</td>
<td>515.426694</td>
<td>528.896401</td>
</tr>
<tr>
<td>processing</td>
<td>543.021943</td>
<td>521.82535</td>
<td>512.190234</td>
<td>525.6791757</td>
</tr>
<tr>
<td>total mem</td>
<td>212M/65M</td>
<td>212M/65M</td>
<td>212M/65M</td>
<td>212M/65M</td>
</tr>
<tr>
<td>processing</td>
<td>0M/2M</td>
<td>0M/2M</td>
<td>0M/2M</td>
<td>0M/2M</td>
</tr>
</tbody>
</table>

Author Name, email@email.com
A Spicy Approach to WebSockets: Enhancing Bro’s WebSockets Network Analysis by Generating a Custom Protocol Parser with Spicy

Bro with Spicy WebSockets parser logging handshake data

`"bro -C -Q -r allpcapsx1000.pcap ws_handshake.evt ws_handshake.bro -s ./ws_handshake.sig"`

<table>
<thead>
<tr>
<th>GNU time output</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>averages</th>
</tr>
</thead>
<tbody>
<tr>
<td>User time (seconds)</td>
<td>1397.17</td>
<td>1424.86</td>
<td>1415.37</td>
<td>1412.46667</td>
</tr>
<tr>
<td>System time (seconds)</td>
<td>300.73</td>
<td>332.82</td>
<td>333.63</td>
<td>322.3933333</td>
</tr>
<tr>
<td>Percent of CPU this job got</td>
<td>118%</td>
<td>117%</td>
<td>115%</td>
<td>134</td>
</tr>
<tr>
<td>Elapsed (wall clock) time m:ss</td>
<td>23:58:54</td>
<td>25:00:90</td>
<td>25:09:50</td>
<td></td>
</tr>
<tr>
<td>Maximum resident set size (kbytes)</td>
<td>335960</td>
<td>335440</td>
<td>336632</td>
<td>336010.6667</td>
</tr>
<tr>
<td>Minor (reclaiming a frame) page faults</td>
<td>91404</td>
<td>91351</td>
<td>91201</td>
<td>91318.66667</td>
</tr>
<tr>
<td>Voluntary context switches</td>
<td>79783093</td>
<td>88103824</td>
<td>89502767</td>
<td>85796561.33</td>
</tr>
<tr>
<td>Involuntary context switches</td>
<td>24553514</td>
<td>31767901</td>
<td>33710834</td>
<td>30010749.67</td>
</tr>
<tr>
<td>File system inputs</td>
<td>256</td>
<td>5768</td>
<td>24</td>
<td>2016</td>
</tr>
<tr>
<td>File system outputs</td>
<td>1282144</td>
<td>1282216</td>
<td>1282176</td>
<td>1282178.667</td>
</tr>
<tr>
<td>Page size (bytes)</td>
<td>4096</td>
<td>4096</td>
<td>4096</td>
<td>4096</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bro -Q initialization</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>averages</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialization</td>
<td>7.778171</td>
<td>9.386094</td>
<td>8.027417</td>
<td>8.397227333</td>
</tr>
<tr>
<td>total time</td>
<td>1438.467763</td>
<td>1500.452032</td>
<td>1509.082221</td>
<td>1482.667339</td>
</tr>
<tr>
<td>processing</td>
<td>1430.689592</td>
<td>1491.065938</td>
<td>1501.054804</td>
<td>1474.270111</td>
</tr>
<tr>
<td>total mem</td>
<td>328M/137M</td>
<td>327M/137M</td>
<td>328M/137M</td>
<td>328M/137M</td>
</tr>
<tr>
<td>processing</td>
<td>0M/2M</td>
<td>0M/2M</td>
<td>0M/2M</td>
<td>0M/2M</td>
</tr>
<tr>
<td>Course</td>
<td>City</td>
<td>Dates</td>
<td>Type</td>
<td></td>
</tr>
<tr>
<td>----------------------------------------------------------</td>
<td>------------------</td>
<td>---------------------------------</td>
<td>------------</td>
<td></td>
</tr>
<tr>
<td>SANS San Francisco Fall 2019</td>
<td>San Francisco, CAUS</td>
<td>Sep 23, 2019 - Sep 28, 2019</td>
<td>Live Event</td>
<td></td>
</tr>
<tr>
<td>SANS Kuwait September 2019</td>
<td>Salmiya, KW</td>
<td>Sep 28, 2019 - Oct 03, 2019</td>
<td>Live Event</td>
<td></td>
</tr>
<tr>
<td>SANS Cardiff September 2019</td>
<td>Cardiff, GB</td>
<td>Sep 30, 2019 - Oct 05, 2019</td>
<td>Live Event</td>
<td></td>
</tr>
<tr>
<td>SANS Northern VA Fall- Reston 2019</td>
<td>Reston, VAUS</td>
<td>Sep 30, 2019 - Oct 05, 2019</td>
<td>Live Event</td>
<td></td>
</tr>
<tr>
<td>SANS Riyadh October 2019</td>
<td>Riyadh, SA</td>
<td>Oct 05, 2019 - Oct 10, 2019</td>
<td>Live Event</td>
<td></td>
</tr>
<tr>
<td>SANS Doha October 2019</td>
<td>Doha, QA</td>
<td>Oct 12, 2019 - Oct 17, 2019</td>
<td>Live Event</td>
<td></td>
</tr>
<tr>
<td>SANS Cairo October 2019</td>
<td>Cairo, EG</td>
<td>Oct 19, 2019 - Oct 24, 2019</td>
<td>Live Event</td>
<td></td>
</tr>
<tr>
<td>SANS Santa Monica 2019</td>
<td>Santa Monica, CAUS</td>
<td>Oct 21, 2019 - Oct 26, 2019</td>
<td>Live Event</td>
<td></td>
</tr>
<tr>
<td>SANS Orlando 2019</td>
<td>Orlando, FLUS</td>
<td>Oct 28, 2019 - Nov 02, 2019</td>
<td>Live Event</td>
<td></td>
</tr>
<tr>
<td>SANS Amsterdam October 2019</td>
<td>Amsterdam, NL</td>
<td>Oct 28, 2019 - Nov 02, 2019</td>
<td>Live Event</td>
<td></td>
</tr>
<tr>
<td>SANS Houston 2019</td>
<td>Houston, TXUS</td>
<td>Oct 28, 2019 - Nov 02, 2019</td>
<td>Live Event</td>
<td></td>
</tr>
<tr>
<td>SANS DFIRCON 2019</td>
<td>Coral Gables, FLUS</td>
<td>Nov 04, 2019 - Nov 09, 2019</td>
<td>Live Event</td>
<td></td>
</tr>
<tr>
<td>SANS Mumbai 2019</td>
<td>Mumbai, IN</td>
<td>Nov 04, 2019 - Nov 09, 2019</td>
<td>Live Event</td>
<td></td>
</tr>
<tr>
<td>SANS Dallas Fall 2019</td>
<td>OnlineTXUS</td>
<td>Sep 23, 2019 - Sep 28, 2019</td>
<td>Live Event</td>
<td></td>
</tr>
<tr>
<td>SANS OnDemand</td>
<td>Books &amp; MP3s OnlyUS</td>
<td>Anytime</td>
<td>Self Paced</td>
<td></td>
</tr>
</tbody>
</table>