About the applicability of Apache2 web server memory forensics

By:
Jan-Niclas Hilgert, Roman Schell, Carlo Jakobs, Martin Lambertz

From the proceedings of
The Digital Forensic Research Conference
DFRWS APAC 2023
Oct 17-20, 2023

Forensic Science International: Digital Investigation 46 (2023) 301610

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

journal homepage: www.elsevier.com/locate/fsidi

DFRWS 2023 APAC - Proceedings of the Third Annual DFRWS APAC

About the applicability of Apache2 web server memory forensics

Jan-Niclas Hilgert *, Roman Schell, Carlo Jakobs, Martin Lambertz

Fraunhofer Institute for Communication, Information Processing and Ergonomics FKIE, Fraunhofer FKIE, Zanderstr. 5, 53177, Bonn, Germany

ARTICLE INFO

Keywords:
Memory forensics
Web servers
Apache

ABSTRACT

With the increasing use of the Internet for criminal activities, web servers have become more and more important during forensic investigations. In many cases, web servers are used to host leaked data, as a management interface for Command and Control servers, or as a platform for illicit content. As a result, extracting information from web servers has become a critical aspect of digital forensics. By default, a lot of information can already be extracted by performing traditional storage forensics including the analysis of logs. However this approach quickly reaches its limits as soon as anti-forensic techniques such as the deletion of configuration files or the deactivation of logging capabilities are implemented. This paper evaluates the feasibility of memory forensics as a complement to traditional storage forensics for cases involving web servers. For this purpose, we present a methodology for extracting forensically relevant artefacts from the memory of Apache web servers, which are among the most commonly used on the Internet. Through various experiments, we evaluate the applicability of our approach in different scenarios. In the process, we also take a closer look at the overall existence of digital traces, which cannot easily be found by following a structured approach. Our findings demonstrate that certain Apache web server structures contain important information that can be retrieved from memory even after the originating event has passed. Additionally, traces such as IP addresses were still found in memory even after complete structures were already overwritten by further interaction. These results highlight the benefits and the potential of memory analysis for web servers in digital investigations.

1. Introduction

In the current digital era, web servers are a fundamental component of the interconnected world we heavily rely on. Due to the increase in cyber crimes in recent years, they are also inevitably utilized by criminals, e.g. for black markets or to provide access to leaked data from ransomware attacks. On the other hand, web servers and web applications have always been a common gateway for attackers to gain unauthorized access to remote computer networks. This fact dramatically increases the involvement of web servers in digital forensic investigations.

Apart from traditional network forensics of captured web server traffic and the analysis of the web server's storage, log files have always been a vital source in investigations and subject of various research in the past. Kumar et al. for example addressed the problem of tampering with log files by proposing a new approach (Kumar et al., 2011), while a more recent paper explored the possibilities of deep learning on server logs (Nazar et al., 2021).

However, the limitations of traditional web server forensics, which primarily relies on log files, were already highlighted in 2017 (Case and Richard III, 2017). The authors pointed out that sensitive information is often not captured in log files, or not logged at all, and that memory forensics could aid in the recovery of valuable data that would otherwise be inaccessible. Despite this, they also noted that existing frameworks lack the capability to automatically extract this data.

In cases where web servers are part of a criminal infrastructure, the servers are often configured to generate no log files. Hence, a vital data source is not available. The same holds for systems where an attacker gained unauthorized access. Clearing the log to conceal the attack is a typical task carried out after successfully gaining access. While avoiding or cleaning log files is easy and a common task of adversaries, tampering with the memory is way more complicated and may even lead to system instabilities when existing software is modified. Moreover, recent studies indicate that tampering with main memory is harder than tampering with evidence on storage media (Schneider et al., 2020).

Our research aims to take a closer look at the possibilities of memory

* Corresponding author.
E-mail addresses: jan-niclas.hilgert@fkie.fraunhofer.de (J.-N. Hilgert), roman.schell@fkie.fraunhofer.de (R. Schell), carlo.jakobs@fkie.fraunhofer.de (C. Jakobs), martin.lambertz@fkie.fraunhofer.de (M. Lambertz).

https://doi.org/10.1016/j.fsidi.2023.301610

Available online 13 October 2023

forensics for web server investigations. For this purpose, we first define the concept of forensically relevant artefacts that are crucial in web server analysis. Then, we delve into Apache2, one of the most widely used web servers, and develop a methodology for an automated extraction of its artefacts from memory. Our proposed methods are put to the test through experiments conducted using a custom framework, which automates various scenarios and generates memory dumps. This evaluation not only covers our structured approach, but also focuses on other remnants that might exist in memory.

2. Artefact creation

Each interaction with a running process can result in a diverse set of artefacts at different places. For example, simply visiting a website with Firefox may lead to updates in multiple databases, new files in a cache folder, and network traffic. Web servers are no exception in that regard. To identify forensically relevant artefacts in web servers' memory, we considered common interactions with these servers and subsequently defined important artefact categories that would be of interest to a forensic investigator during their analysis. For a better understanding, the resulting artefact categories will be presented first.

2.1. Artefacts

Configuration: The configuration of a web server is a critical artefact in any forensic investigation, as it provides valuable information that helps to direct the focus of the investigation. This includes details about the defined hosts on the web server, such as their IP addresses and ports, as well as paths to the resources served by the server, which can assist in analyzing its storage.

Connection: Undoubtedly, information about connections involving the analyzed web server is one of the most important artefacts in any investigation. This includes details such as IP addresses of remote clients that connected to the server, ports used, and timestamps, if available. This information can be incredibly valuable in filtering and analyzing large amounts of network traffic and in identifying other systems that may play a crucial role in the investigation. It is important to note that successful connections without any exchanged requests may not be recorded in common logs, making the extraction of this information from the web server's memory a crucial aspect of the forensic process.

TLS data: Most web servers on the market make use of existing TLS libraries to handle encrypted connections. For this reason, artefacts specific to these libraries (e.g. key material) are not considered a web server artefact themselves. However, this category refers to all artefacts that are created by the web server and contain information about a TLS connection or possibly the server certificate and private key.

Requests & Responses: In addition to the sole information of the existence of a connection, it is in many cases helpful to know what kind of communication occurred. In a first step, this includes artefacts about the request that was made by the client as well as the response that was provided by the web server along with their corresponding headers. These artefacts can already convey important information, for example in the case of a brute force attack or the use of GET parameters.

Content: In a second step, the content transferred between the client and web server contains even more insights into a past connection. While static resources can usually still be extracted from the server, this artefact is especially important in cases of dynamic content, which cannot be retrieved by a simple file system analysis.

2.2. Interactions

Start a web server: This interaction is in fact mandatory for all running web servers. After performing this interaction, it is expected that information about its current configuration can be extracted. In case of a server supporting TLS connections, this may also include information of the TLS category.

Reload web server: It is possible to encounter a web server, whose configuration has been reloaded without restarting the process itself. Expected artefacts are naturally the newly loaded configuration, but may also include remnants of the previous configuration.

Connect: Prior to any interaction with a client, a connection must be established. This can also be the case for port scanning attacks, in which case no further content is sent. Naturally, this involves artefacts from the connection category.

Receive request: Receiving a request is the most common interaction to occur on a web server. This will not only create artefacts about the request and its content, but likewise the sent response of the server. Furthermore, this interaction results in connection information of the server. In the case of a TLS connection, also TLS related data can be expected.

Send a response: While this is not a direct interaction with the server, it is an implicit reaction to the reception of a request. Sending a response also results in similar artefacts about the response like headers and status codes, but can also result in content that may be loaded into memory before it is sent to the client.

3. Artefact extraction

This section describes our proposed methodology for the extraction of the most critical Apache2 web server artefacts. These methods can be applied to any previously obtained full as well as process memory dump – provided that virtual addresses are handled correctly, e.g. by translating them to physical offsets within the memory dump itself.

Since developing methodologies for finding forensic artefacts in memory reliably can be a tedious task, we summarize the basics concepts that we utilized for our approach:

- **Documentation & Source code:** Having access to the source code of the application of interest is without any doubt an enormous benefit when looking for promising artefact sources. For our research, we made use of the available Apache2 source code as well as its documentation to identify structures of high forensic value (Apache2, 2023).
- **Hard coded values:** Finding such structures in memory however can become infeasible, if no suitable unique characteristic of the structure exists. A rewarding example of such a characteristic are hard coded values within the structure itself. These values act similar to magic bytes in headers or footers of files and provide a good starting point for the detection of a certain structure. This can also be the case for special instances of a structure as shown in the example below.

```
APR_DECLARE(void) apr_pool_tag(
apr_pool_t *pool, const char *tag)
{
    pool->tag = tag;
}
```

Listing 1. Method in `apr_pools.c` used to set the `tag` field of a given `apr_pool_t` structure.

```
    apr_pool_create(&process->pconf,
process->pool);
    apr_pool_tag(process->pconf, "pconf");
```

Listing 2. Part of `httpd/server/main.c` creating a pool and hard-coding its name tag to "pconf".

- **Value ranges:** Besides hard coded values, some members of a structure may (or at least should) only take values from a limited range. In the C programming language, this is often the case when enumeration data types are used. The following code example shows the definition of the `ap_conn_keepalive_e` data type, which is for example used in the `conn_rec` structure described later on.

```
typedef enum {
    AP_CONN_UNKNOWN,
    AP_CONN_CLOSE,
    AP_CONN_KEEPALIVE
} ap_conn_keepalive_e;
```

Listing 3. Declaration of enumeration data type `ap_conn_keepalive_e` in `httpd/include/httpd.h`.

- **Pointer searches:** Most structures in C utilize pointers, which reference strings and other dynamically sized members. A pointer can then be used to find the location of the member it points to in memory. On the other hand, if the location of a member is known, its address can be used to search for any possible pointers pointing to that specific member. This way it is possible to walk structures backwards.
- **Sanity Checks:** It is important to implement sanity checks while extracting data from memory to reduce the likelihood of false positive results. For instance, a member representing a port number should have a value that falls within the range of 0–65535. These checks ensure that the extracted data is reliable and accurate.

3.1. Configuration

Apache makes use of a `server_rec` structure, which contains the most important information about each virtual server that has been configured. This does not only include fundamental values such as the server's host name, listening address and port, but also paths to the error logs and the configuration file used to define this virtual server along with the exact line number.

The task of finding a `server_rec` structure within memory is complex, as many of its members' values are dependent on the server's configuration and are thus not suitable for a direct search of the structure. To address this, we have devised a three-step process for the extraction of the `server_rec` structure, based on the links between multiple structures as depicted in Fig. 1.

1. The `server_rec` structure starts with a pointer to a `process_rec` structure. As the name suggests, this structure describes the process, in which the virtual server is running. The `process_rec` structure itself contains only little information. However, also the arguments provided to the process may be of relevance for an investigation and should thus be extracted as well.
2. To find a `process_rec` structure in memory, we leverage its second attribute, which is a pointer to the configuration pool, referred to as `pconf`. These `apr_pool_t` structures are used for the management of memory regions.
3. For the detection of the `pconf` pool, we take advantage of its `tag` attribute, which is a pointer to a string, defining the tag of a pool. In the case of the pconf memory pool, the tag is simply "`pconf`".

Thus, we search for all `pconf` strings in memory and traverse backwards until we reach a potential `server_rec` structure, which is then validated and parsed to extract all relevant configuration information. For multiple defined virtual servers, multiple of these structures are used and present in memory.

3.2. Connection

Information about connections handled by the web server is stored in the `conn_rec` structure. It contains pointers to the IP addresses of the client and the local server. Furthermore, for each of these it contains a pointer to a `apr_sockaddr_t` structure defined by the Apache Portable Runtime library utilized by the web server. As Fig. 2 illustrates, this structure stores additional information such as the ports used.

For the detection of `conn_rec` structures in memory we leverage its second member, a pointer to the `server_rec` structure of the virtual host the connection belongs to. After we identified all of the available server structures in the previous step, we can use their virtual addresses to search for possible `conn_rec` candidates. Since other structures may also store the same pointer, we employed additional information for our search in order to reduce the number of false positives.

The `keepalive` member of the structure stores, whether the connection should be kept alive for a future request. As demonstrated in the previous section in Listing 3, the value of this member can only take three possible values: 0, 1, or 2. Another of its members with a limited set of values is `outgoing` indicating the direction of the connection. As shown in Listing 4, a comment in the source code reveals that a valid value can only be 0 or 1 at the moment.

```
    /* Some day it may be flags, so deny anything but
    0 or 1 for now */
    if (outgoing > 1) {
        return NULL;
    }
```

Listing 4. Comment in `server/connection.c` describing the intended values for the `outgoing` member of `conn_rec`.

By including arbitrary place holders for our search pattern, this information can be combined to specifically search for `conn_rec` structures in memory. Additionally, sanity checks can be applied by validating values such as the `port`, `family` or `ipaddr_len` within corresponding `sock_addr_t` structures.

3.3. Request

One of the most forensically interesting structures used by the

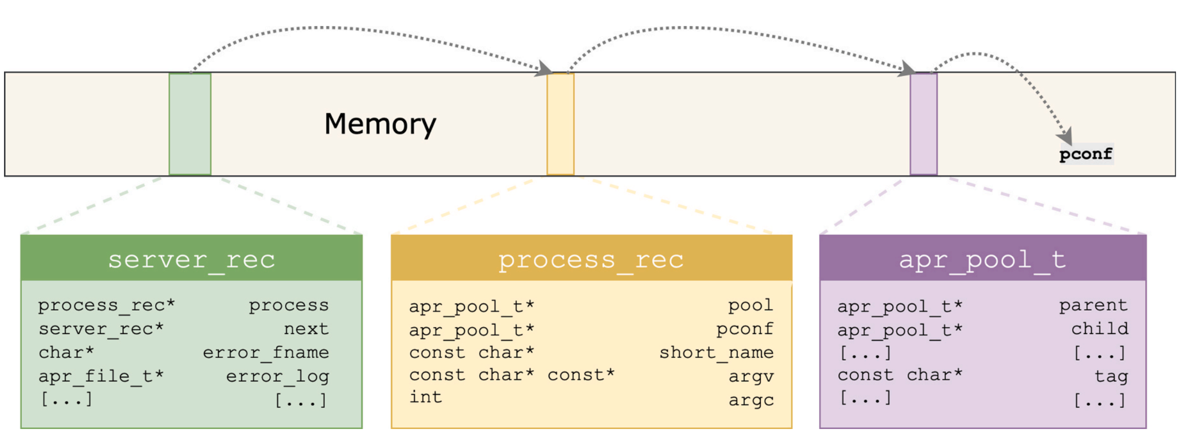

Fig. 1. Methodology for the extraction of a `server_rec` structure.

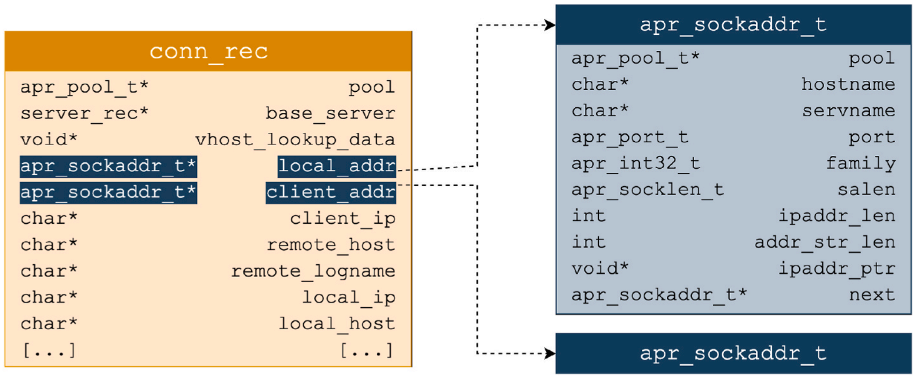

Fig. 2. Connection information stored in `conn_rec` and `sock_addr_t` structures.

Apache web server is the `request_rec` structure representing a received request. As depicted in Fig. 3 the structure contains a pointer to the connection it originated from as well as a pointer to the virtual host handling this very request. Both of them are represented by a `conn_rec` and `server_rec` structure respectively. Furthermore, the structure contains multiple pointers to various specific parts of the request, such as the protocol, hostname or URI. Additionally, certain values like the time of the request or other integers are contained directly within the structure.

Apache2 does not have a dedicated structure for the responses it generates. Instead, a lot of the information regarding a response to a certain request is stored in the corresponding `request_rec` structure. This includes a pointer to the status line as well as the status code encoded as an integer, but also all of the headers of the response itself. Both, the headers of the response as well as the headers of the request are stored using an `apr_table_t` structure. For parsing the contained information it is only necessary to access its first member, which is a `apr_array_header_t` structure defining the number of elements in the list along with the corresponding size of an element as shown in Fig. 4. Using this information it is possible to access all of the individual elements of the table. In e table. In our example, each element corresponds to a key:value pair of a header.

To detect the `request_rec` structures in memory, we leverage specific members that should have values within a well-defined range. These members include:

- `proto_num`: This integer-typed member indicates the HTTP protocol version used in the request, such as HTTP 1.1, which is represented by `1001` or `0x03e9` in hexadecimal. We include values for all available HTTP versions in our search.
- `status`: Also an integer, this member provides information on the status code of the response, such as `0x0194` for a 404 Not Found response and `0x00c8` for a 200 Found response. Our search includes the values for the most common HTTP status codes.
- `proxyreq`: This enumerated datatype member is used to denote proxy requests and can only have values of 0, 1, 2, or 3.

Considering these limitations for some of its members eliminates the necessity to detect other structures for the extraction of `request_rec` structures from memory beforehand. However, a previously detected `server_rec` structure could be used for validation.

3.4. Content

Recovering remnants of content transmitted by the server and, especially, by the client can be a valuable asset in an investigation. However, the most effective method for extracting this content from memory is contingent upon the way in which the data was processed by various modules. As a result, we concentrate our efforts on extracting the smallest units utilized by Apache2's memory management. While this approach guarantees that a significant portion of data can be recovered in theory, it also means that some data may still need further interpretation.

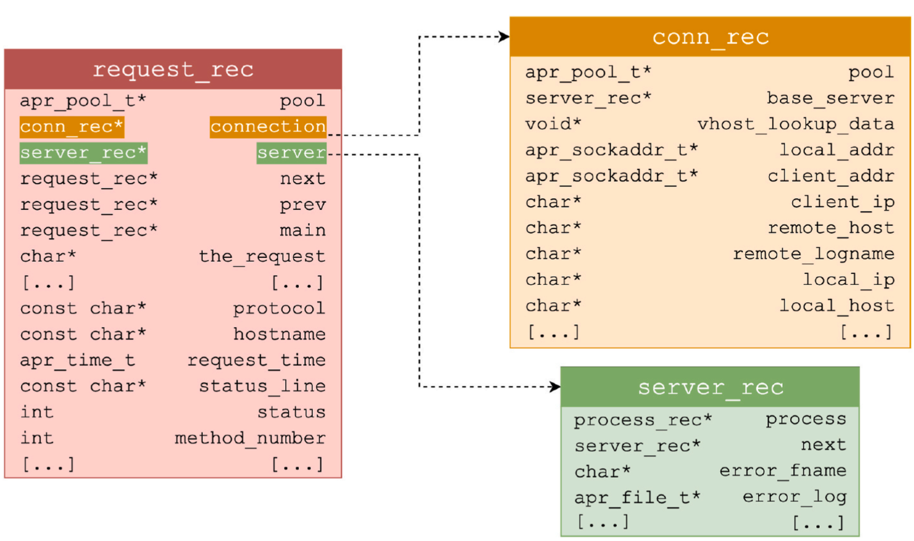

Fig. 3. Structure of a `request_rec` used in Apache2.

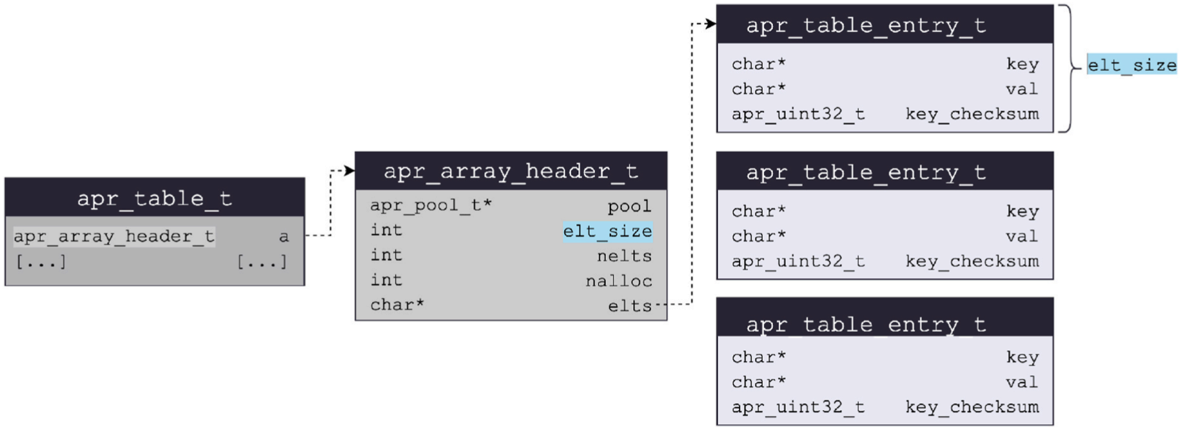

Fig. 4. Links between an `apr_table_t` and its entries.

```
/* All of the bucket types implemented by the core */
1258 [...] apr_bucket_type_flush;
1264 [...] apr_bucket_type_eos;
1268 [...] apr_bucket_type_file;
1273 [...] apr_bucket_type_heap;
1278 [...] apr_bucket_type_mmap;
1285 [...] apr_bucket_type_pool;
1289 [...] apr_bucket_type_pipe;
1295 [...] apr_bucket_type_immortal;
1301 [...] apr_bucket_type_transient;
1305 [...] apr_bucket_type_socket;
```

Listing 5. Bucket types used by Apache2 defined in `apr/include/apr_buckets.h`.

Apache2 makes use of so called *buckets* to store various kinds of data. A bucket is never used alone, but instead multiple buckets are grouped together and stored in a ring structure referred to as a *bucket brigade* (Apache Tutor). Listing 5 provides an overview of the defined bucket types, which can not only refer to data in memory (e.g. `apr_bucket_type_heap`), but also to a part of a file (i.e. `apr_bucket_type_file`). Since brigades and buckets are used to hold content data as well, our methodology describes how to extract bucket structures.

As shown in Fig. 5, each `apr_bucket` structure starts with a pointer to a bucket type structure. For each type of bucket, a separate type structure is declared and stored in memory. An example for such a declaration can be seen in Listing 6. We can identify a specific type structure by utilizing its `name` attribute. In our example, the heap and file bucket types store a pointer to the strings "`HEAP`" and "`FILE`", respectively. Similar to our previous `server_rec` approach, we can search the memory for all occurrences of these strings and use their addresses as a search parameter for the corresponding `apr_bucket_type_t` structure.

To minimize the number of false positives, we also make use of the hard-coded number of functions, which is 5 in our example. Once the virtual address of a specific bucket type structure has been found, it can be used to detect all buckets of that type, represented by `apr_bucket` structures pointing to that specific type structure. Each type of bucket has its own structure, and in the case of a heap bucket, the final data stored in the bucket is referenced by the `base` pointer.

```
APR_DECLARE_DATA const apr_bucket_type_t
    apr_bucket_type_heap = {
    "HEAP", 5, APR_BUCKET_DATA,
    heap_bucket_destroy,
    heap_bucket_read,
    apr_bucket_setaside_noop,
    apr_bucket_shared_split,
    apr_bucket_shared_copy
};
```

Listing 6. Instance for a heap bucket type defined in `apr/buckets/apr_buckets_heap.c`.

3.5. TLS data

Our research concentrates specifically on the extraction of artefacts unique to the Apache2 web server. As such, the extraction of information from structures created by cryptographic libraries like OpenSSL falls outside the scope of our investigation. There has already been

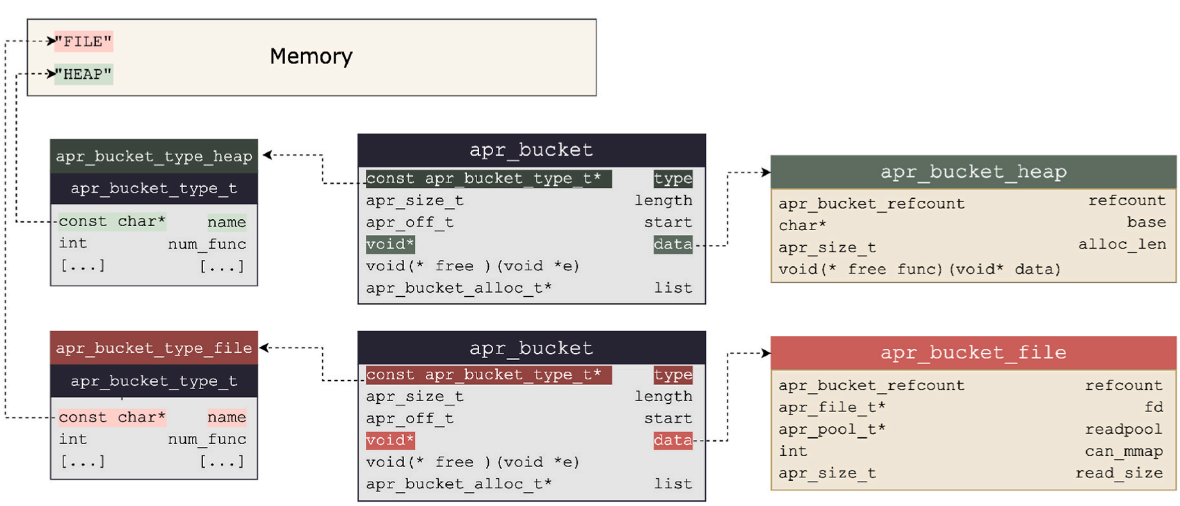

Fig. 5. Structures for two buckets of type `heap` and `file`.

extensive research on this topic, with several publications addressing the challenge of extracting key material from the memory of these libraries (Maartmann-Moe et al., 2009; Taubmann et al., 2016). Nevertheless, having an understanding of the links between Apache2's own structures and external libraries is useful and can be leveraged to apply other methods focused on TLS-specific structures.

3.5.1. Configuration

As previously discussed, the `server_rec` structure contains crucial information regarding the configuration of a virtual host. Some of this information, such as the document root, is stored in a `core_server_config` structure. A pointer to this configuration structure can be located as the first element in a configuration vector, which is referenced by the `module_config` attribute in the server structure, as illustrated in Fig. 6. This configuration vector holds pointers to various modules' per-server configuration structures. When SSL is enabled, our experiments have shown that the corresponding SSL configuration structure, `SSLSrvConfigRec`, can typically be found at index 26, though this value may vary.

The discovered SSL configuration structure links to the `modssl_ctx_t` structure, which in turn points back to the SSL configuration. By leveraging this relationship, it is possible to validate the candidates in the configuration vector, in case the correct module index is not known. Moreover, the `modssl_ctx_t` contains a pointer providing a crucial link to a SSL_CTX object, which is used by OpenSSL to establish secure SSL/TLS connections (OpenSSL Foundation). Furthermore, it also holds a pointer to the `modssl_pk_server_t` structure, which can be used to obtain the file paths of the server's certificate and key files, providing important information for the investigation.

3.5.2. Connection

Apache2 makes use of a `SSLConnRec` whenever a TLS connection is handled. Besides a pointer to an `SSL` structure, it also contains a reference to a potential client certificate and used cipher suites. Since it also contains a pointer to the structure of the server handling the connection, we leverage its previously known address to search for any `SSLConnRec` candidates. For this search, we also exploit the fact that the corresponding pointer to the server structure is preceded by members with a limited range of values, such as `non_ssl_request`. This enumeration datatype can only take values of 0, 1, 2 and 3.

4. Evaluation

4.1. Dataset creation

For the development and evaluation of our methodologies, we created a custom framework that can establish a running web server and execute a predefined set of actions, such as requests. The framework then captures the memory dump of the web server's processes by iterating over the regions using the current process information stored in `/proc/<PID>/maps` and `/proc/<PID>/mem`. Note that this approach loads all requested data into memory, which might not be the case when obtaining a full physical memory dump. However, the corresponding data can then be found on the persistent storage (e.g. swap files), which should be included during the investigation. Furthermore, our framework also extracts additional data such as its log files or configuration. It also records a network capture of all interactions with the server including potential TLS keys, which, when combined with the other data, creates a reliable ground truth for our evaluation.

Both the web server and the clients used to perform requests were implemented using Docker containers. This approach not only optimizes resource usage but also offers the flexibility to test various web server solutions in the future. The actions performed during the experiment are defined in a YAML file, making it easy to share and reproduce the scenarios. The framework as well as the YAML configuration files used in the subsequent evaluations are publicly available (Hilgert et al., 2023).

4.2. Experiments

In this section, we present a series of experiments we have conducted in order to create as well as recover forensically relevant artefacts from Apache2 memory dumps. Our Apache2 setup employs the `event` Multi-Processing Module (MPM), which is based on the `worker` MPM (The Apache Software Foundation). This configuration involves a parent process that spawns child processes, which in turn create server and listener threads. In our experiments, we use the default configuration, which spawns two child processes, resulting in a total of three Apache2 processes whose memory dumps will be acquired and analyzed in the course of the experiments. Recovery of artefacts was performed by our automated extraction method following a structured approach as well as by searching for known traces in memory directly. This was done to showcase what artefacts actually remain in memory.

4.2.1. Configuration

For the purpose of this evaluation, we set up a simple server configuration as depicted in Fig. 7. The configuration specifies a virtual host that listens on port 443 and all IP addresses, represented by the asterisk. The server name used is `example.com`. The right side of the figure displays the results of our automated artefact extraction from memory, following the methodology outlined in the previous section. As it can be seen, all the necessary information was extracted successfully, including the document root of the virtual host, as well as the path and precise line number of the virtual host's configuration. This information

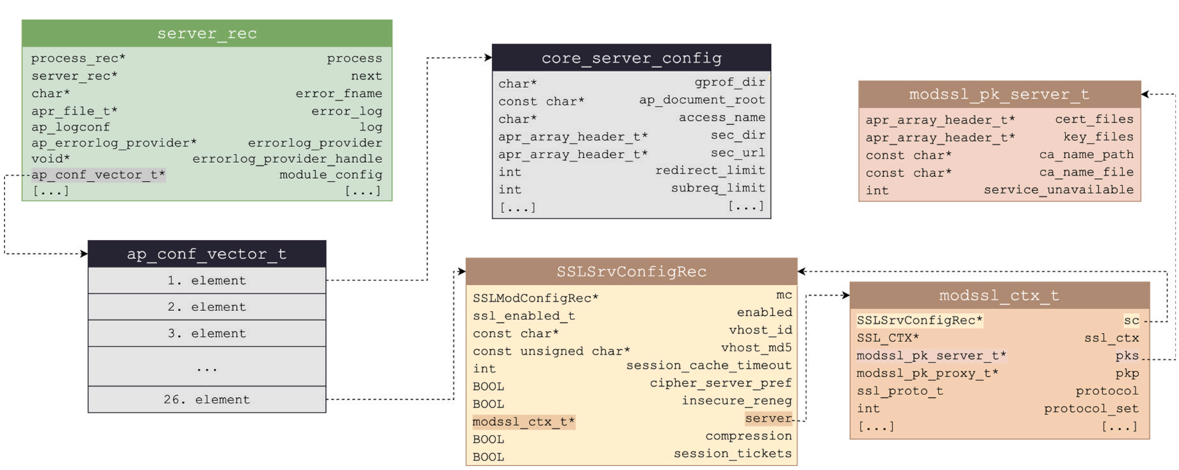

Fig. 6. Link between the `SSLSrvConfigRec` and `server_rec` structures.

```
$ cat example.com.conf
	# vim: syntax=apache ts=4 sw=4 sts=4 sr noet

	<VirtualHost *:8081>
		ServerName example.com
		ServerAlias www.example.com

		ServerAdmin webmaster@localhost
		DocumentRoot /var/www/example.com/

		ErrorLog ${APACHE_LOG_DIR}/error.log
		CustomLog ${APACHE_LOG_DIR}/access.log combined

	</VirtualHost>
```

```
$ ./extract_server_information.py apache2/dumps
[...]
	error_fname -> /var/log/apache2/error.log
	module_config ->
		[...]
		ap_document_root -> /var/www/example.com
	defn_name -> /etc/apache2/sites-enables/example.com.conf
	defn_line_number: 3
	is_virtual: 0x255
	server_admin -> webmaster@localhost
	server_hostname -> example.com
	server_addr_rec ->
		virt_host -> *
		host_addr ->
		[...]
		port: 8081
[...]
```

Fig. 7. Content of `example.com.conf` configuration file compared to artefacts extracted from memory by our structured approach.

is particularly critical in scenarios where the original configuration file has been deleted or altered on the persistent storage. The corresponding `server_rec` structure was present in all three of the acquired process dumps.

4.2.1.1. Reload a configuration. In this experiment, we investigated the possibility of retrieving remnants of virtual hosts in memory even after the web server has reloaded a new configuration file. To test this scenario, we set up an Apache2 web server and defined ten different virtual hosts, ranging from `example-one.com` to `example-ten.com`, each listening on a different port between 8081 and 8090. Afterwards, we reloaded the server with a new configuration that only defined one virtual host for `example-reloaded.com` on port 8091. Finally, we checked if any `server_rec` structures of the previously defined virtual hosts could still be found in memory.

In our experiments, a reload of the Apache2 web server, initiated by the command `service apache2 reload`, resulted in the replacement of the two running child processes with new processes. Regarding `server_rec` structures, only the structure for `example-reloaded.com` could be found in the process memory of all three running Apache2 processes. All other matches appeared to be false positives. Nonetheless, multiple traces of the previously defined virtual hosts were still present in the process memory, as depicted in Fig. 8. Our test results revealed that this included, for instance, all of the 10 previously defined server host names.

4.2.2. Connection

This experiment focuses exclusively on the artefacts created by a basic connection, which we define as a *successful TCP handshake* initiated by a client. To accomplish this, a predetermined number of

```
5380h  3c2f 5669 7274 7561 6c48 6f73 743e 006f  </VirtualHost>.o
5390h  c826 3db5 aaaa 0000 d893 b6a7 ffff 0000  .&=.............
[...]
53d0h  0000 0000 0000 0000 2a3a 3830 3832 3e00  ........*:8082>.
53e0h  f031 3db5 aaaa 0000 2894 b6a7 ffff 0000  .1=.....(.......
[...]
5420h  0000 0000 0000 0000 6578 616d 706c 652d  ........example-
5430h  7477 6f2e 636f 6d00 7833 3db5 aaaa 0000  two.com.x3=.....
[...]
5470h  2900 0000 0000 0000 0000 0000 0000 0000  )...............
5480h  7777 772e 6578 616d 706c 652d 7477 6f2e  www.example-two.
5490h  636f 6d00 ffff 0000 b031 3db5 aaaa 0000  com......1=.....
[...]
54d0h  2b00 0000 0000 0000 0000 0000 0000 0000  +...............
54e0h  7765 626d 6173 7465 7240 6c6f 6361 6c68  webmaster@localh
54f0h  6f73 7400 0000 0000 c02c 3db5 aaaa 0000  ost......,=.....
[...]
5530h  2c00 0000 0000 0000 0000 0000 0000 0000  ,...............
5540h  2f76 6172 2f77 7777 2f65 7861 6d70 6c65  /var/www/example
5550h  2d74 776f 2e63 6f6d 2f00 b6a7 ffff 0000  -two.com/.......
5560h  1833 3db5 aaaa 0000 a895 b6a7 ffff 0000  .3=.............
```

Fig. 8. Traces of previously defined virtual hosts in memory.

connections were consecutively established at three-second intervals by different clients, each with a unique IP address. Each connection was ended prior to the establishment of the next one, so that no active connections were present when the memory dump was taken.

Regardless of the number of established connections, only two of the `conn_rec` structures could be recovered from memory matching the number of child process created in our experimental setup. For this reason, we additionally evaluated the pure existence of the client IP addresses involved in the previous connections within the acquired memory dump. The results for different numbers of connections were:

- **25 connections:** All of the IP addresses were found *twice* within each of the three Apache memory dumps.
- **50 connections:** All of the IP addresses were found *twice* within each of the three Apache memory dumps except for the first.
- **100 connections:** In this scenario, the IP addresses of early connections were not found, while the IP addresses of later connections were present in memory. The availability of IP addresses for connections in the middle was inconsistent, which prompted us to repeat the experiment 50 times and count the overall number of IP addresses found. The results, shown in Fig. 9, indicate that roughly the last 50 addresses could be found at least 300 times (which equals two IP addresses per process dump times 50 runs), while the first addresses were only found rarely or not at all. The results for the middle section vary. Furthermore, we also counted the number of correctly extracted connection structures within these 50 runs. It was observed that they were only present for the most recent requests.

The results of the experiment indicate that a significant number of IP addresses from past connections can still be present in memory. This is particularly valuable in forensic investigations, as these IP addresses are not recorded in any log files.

4.2.3. Requests & responses

In the first experiment, we generated a basic HTTP GET request to retrieve an image from our web server. After the server sent the response and closed the connection, we obtained a memory dump. The results, as depicted in Fig. 10, reveal that important information from the request, such as connection details, the header of the request, and the complete path to the requested resource, could still be recovered from memory by searching for `request_rec` structures. As previously noted, Apache2 does not have a dedicated structure for responses, so information about the response, such as the status code and headers, is also stored in the request structure and was successfully extracted.

Persistence of request artefacts

In another experiment, we focused on the persistence of requests in memory similar to the prior experiment involving connections. This time, 100 subsequent requests were made by different clients. Our aim was to determine not only the presence of any `request_rec` structures

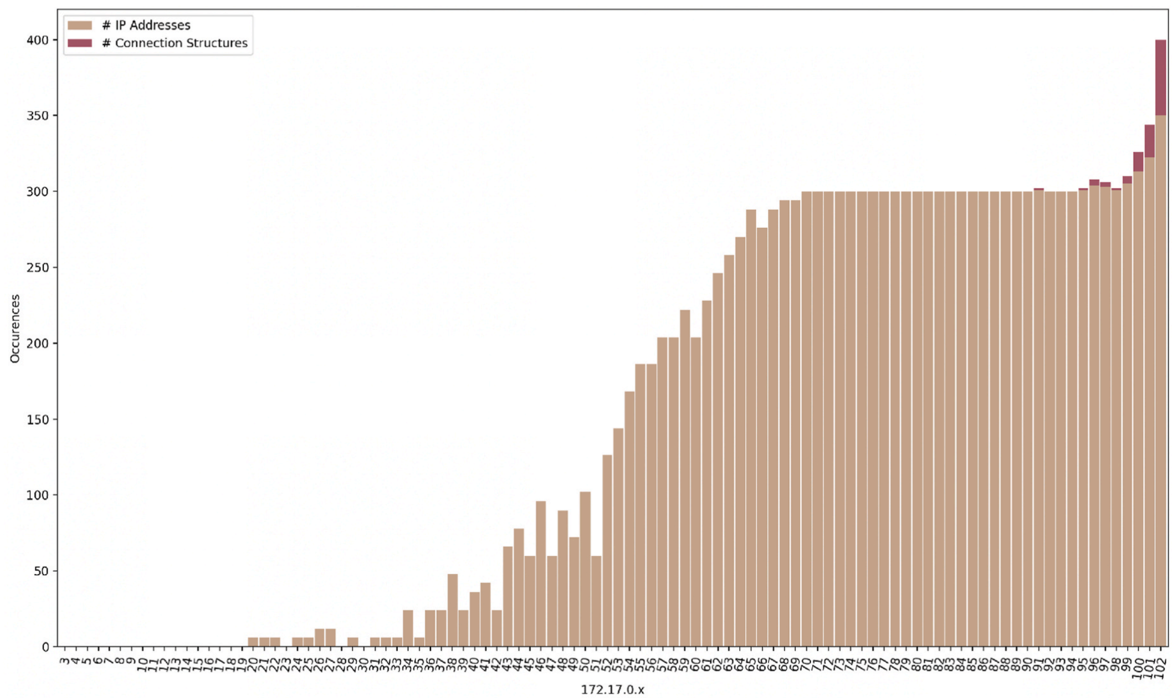

Fig. 9. Cumulative occurrences of `conn_rec` structures and IP addresses within memory after 50 iterations, 100 connections each.

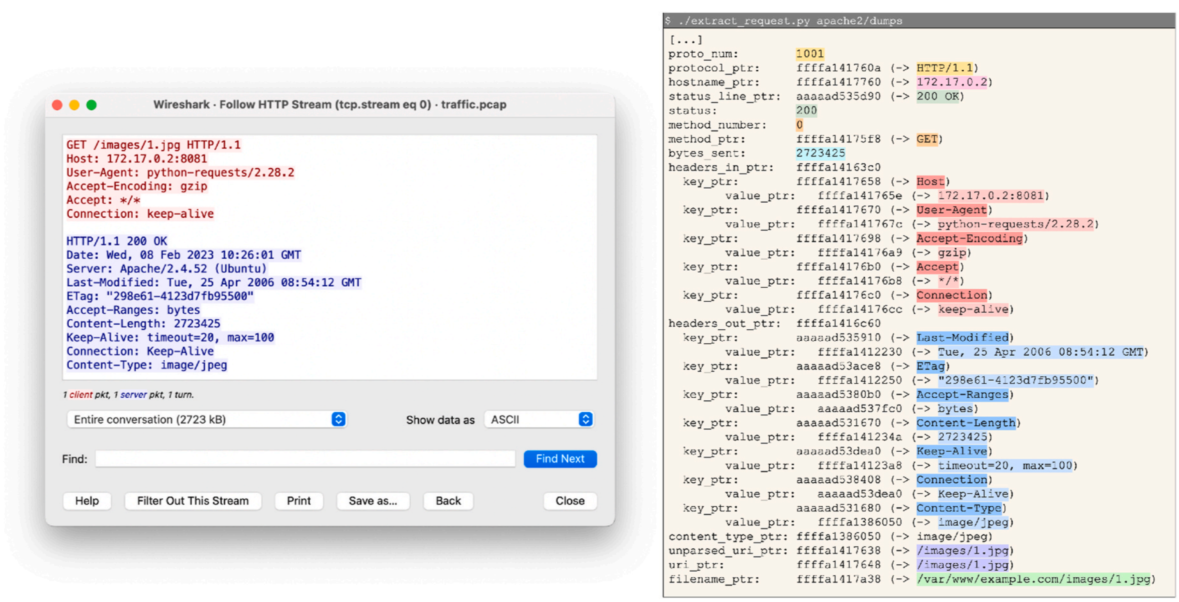

Fig. 10. Actual request in Wireshark compared to the extracted information from memory.

still present in memory, but also to assess the pure existence of any information about previous requests. To do this, we conducted a further search for complete request lines, which consist of a method, URI, and HTTP version number. As each client requested a unique resource, we were able to match the found request lines in memory to the corresponding client. This experiment was repeated 50 times to obtain a comprehensive understanding of the persistence of requests in memory. Furthermore, we also attempted to identify any strings created for logging purposes in memory to distinguish them from other request remnants.

Fig. 11 shows that similar to the previously extracted connection structures, request structures could only be found for later requests. It can also be seen that there is a higher likelihood of finding request lines from more recent requests in memory. However, in some cases even request lines from very early requests could still be found. We also observed that the amount of artefacts left in memory by different requests was inconsistent, with some requests leaving significantly more trace than others. Despite a manual examination of the resources and memory dumps, no clear explanation for this disparity was found.

Although the number of retrievable request structures from memory is limited, information about prior requests remains accessible. The ability to search for a complete request line, made up of a method, URI, and HTTP version number, is a valuable tool for detecting this information. In real-world scenarios, the exact request line may not be known, but by exploiting the limited possible values for methods and HTTP version numbers, it is possible to create a search pattern for the

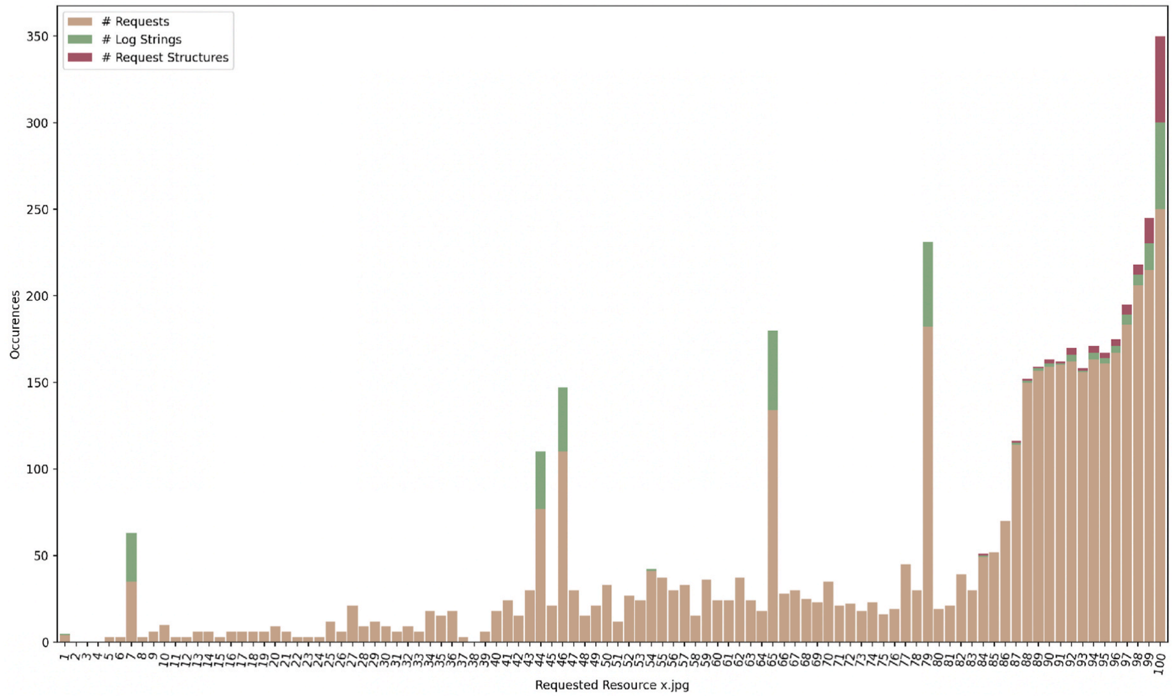

Fig. 11. Remnants of requests within memory after 50 iterations, 100 requests each.

detection of HTTP requests.

4.2.4. Content

Before we evaluate the structured extraction of the previously mentioned *buckets* used by Apache, it is crucial to asses the amount of content that could at most be retrieved from memory. For this reason, we conducted experiments to show, which parts of a sent or received file actually exist in memory after the request was handled. For this, we created an artificial file with a unique pattern, which enabled us to know the exact original location of any detected fragment of the file in memory. We created artificial files in sizes of 5 MB, 50 MB and 500 MB, which were stored on the server and requested by a client. Furthermore, the files were transferred in different ways including the use of TLS and compression as shown in Table 1.

It can be seen that for a transmission without TLS and in which the resource was not compressed, no traces of the file were found in memory. This could be to various reasons, one of them being the `EnableSendfile` feature in Apache2, which bypasses the necessity for files to be loaded into memory when they are sent. When gzip was used as a compression algorithm, parts of the file were existent in memory, since it had to be loaded into memory for compression. The same holds true for TLS connections. However, the amount of data that could be found in memory was very limited compared to the original size of the requested file.

When running these experiments multiple times we observed irregularities in the amount of content that was present in memory. These scenarios are marked with an * in Table 1, which gives the most frequent (in our cases in more than 80% of all runs) value that could be observed. Deviations of these values were highest for the gzip scenarios of the 50 MB file, in which in some instances up to roughly 4 MB of the artificial file could be found in memory.

POSTed data

In this experiment, we concentrate on data transmitted by the client, which is accomplished by sending a single POST request with similar artificial files to the server. Since POSTed data is in many cases subject to further processing by other processes, we proxy the request to a simple Flask application in the background utilizing the `mod_proxy` module. The Flask application simply accepts the request and returns a 200 OK response.

Table 2 shows the results for varying data sizes, taking into account that POSTed data by clients is usually smaller, e.g. when sending form data. The data was always sent uncompressed. It can be seen that except for the smallest file without TLS, remnants of each resource could be found in memory, even when no further processing by compression or cryptographic libraries occurred. When TLS was used, it was even possible to detect almost the complete data of the 5 kB file.

Fig. 12 shows, which parts of the 5 MB files could be detected in memory. Each block in the map represents a 512-byte fragment of a transferred file. The color of a block indicates how often it has been found in memory summed up over all 50 experiment runs. Surprisingly, the data that was still present in memory was not the exact end of the file and the data areas were rather stable. There is one area in the middle of the 5 MB, which we found reliably in almost all experiments. Furthermore, the beginning of the file as well as a different area in the middle could be found in roughly half of the cases. The same behavior was also observed for the 5 MB file sent without TLS.

Table 1
Amount of bytes of a requested resource that could be found in memory after the request.

Sent via		**5 MB**	**50 MB**	**500 MB**
No TLS	Plain	0 kB	0 kB	0 kB
	gzip	33.28 kB	33.28 kB*	4259.84 kB*
TLS	Plain	18.94 kB	18.43 kB*	13.82 kB*
	gzip	33.28 kB	33.28 kB*	4259.84 kB*

Table 2
Amount of bytes of a sent resource that could be found in memory after the request.

Sent via		**512 B**	**5 kB**	**5 MB**
Plain	No TLS	0 kB	1.2 kB	9.73 kB
	TLS	0.5 kB	5 kB	24.57 kB

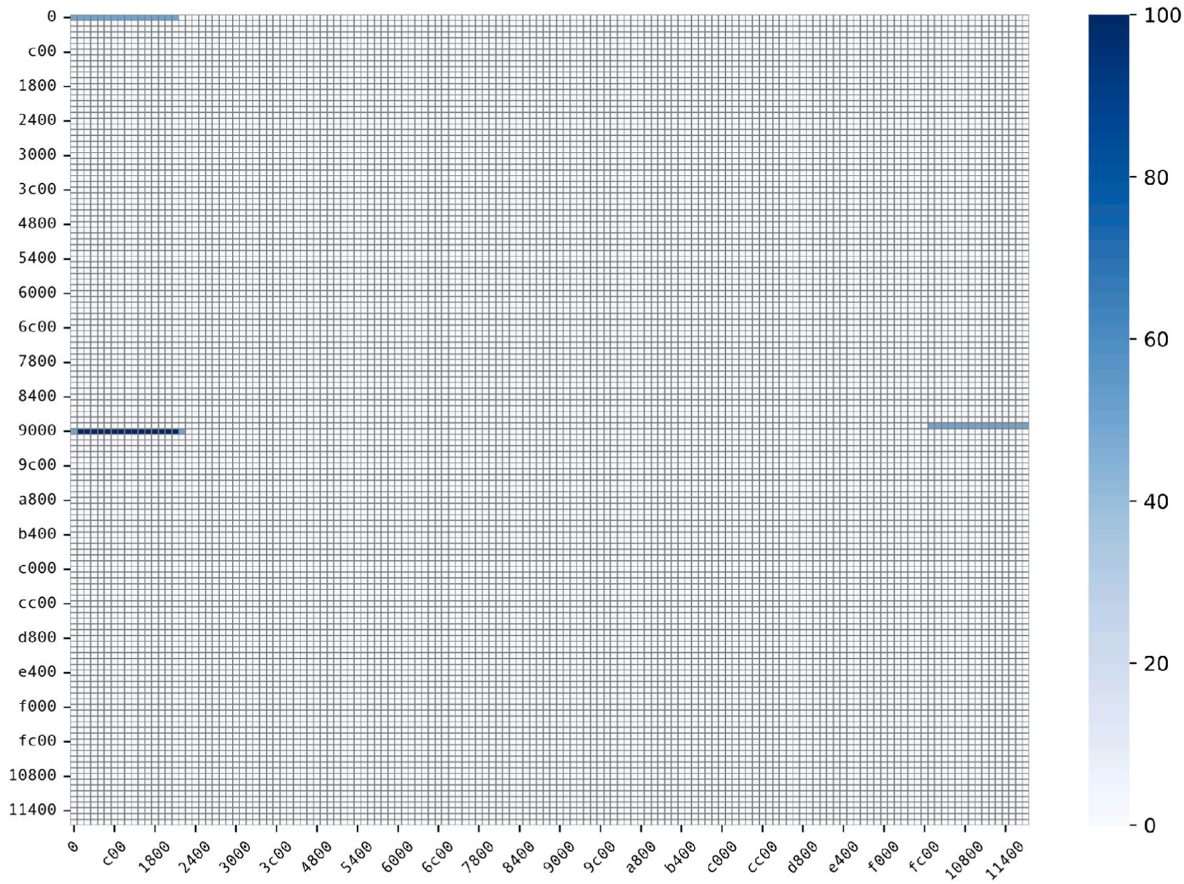

Fig. 12. Presence of 512 Byte blocks of the 5 MB files sent via POST requests with TLS in memory.

Extraction of buckets

In our evaluation, we focused on the feasibility of extracting heap buckets for the POST scenarios described earlier. Surprisingly, only two valid heap buckets were identified in the memory of the six cases analyzed. One of these buckets contained the response data sent by the web server, as depicted in Fig. 13. This was observed even in the cases where TLS was used. However, the origin of the data inside the second bucket remains unknown and requires additional investigation. Moreover, by employing our methodology for extracting heap buckets, other types of buckets can also be extracted to assess their usefulness in investigations in the future.

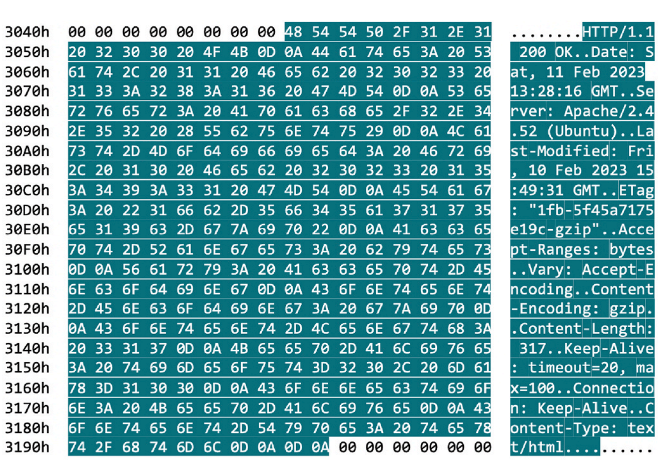

Fig. 13. Extracted heap bucket data containing the HTTP response.

4.2.5. TLS data

As described earlier, we only focus on Apache2 specific artefacts. For this reason, artefacts creating any crucial key material were not found. In a first experiment, we evaluated our approach for the extraction of a TLS configuration. The results, as shown in Fig. 14, demonstrate that it is possible to determine if TLS is enabled for a virtual host and retrieve the paths for the certificates and keys utilized by the server. Furthermore, the extracted configuration contains a valuable link to the `SSL_CTX`.

We also applied our methodology for the extraction of `SSLConnRec` structures. Unfortunately, this approach provided limited value as the only information that could be extracted, besides the pointer to the `SSL` member, was the cipher suite, which was stored in the format described by OpenSSL (e.g. `HIGH:!aNULL:!aNULL:!eNULL:!EXP`) (OpenSSL Foundation, Inc.).

4.2.6. Robustness

The previous experiments were performed on Apache version 2.4.52, which is the default package installed via `apt` on Ubuntu 22.04 at the time of writing this paper. Since structures may change over time, we chose to evaluate the robustness of our approach, by testing the extraction of relevant artefacts on other versions of the Apache web server. According to recent statistics, Apache 2.4 is currently the most prevalent version (W3Techs, 2023). However, 10% of websites running Apache are still using version 2.2, whose life time already ended in 2018. For this reason, we performed additional experiments on Apache 2.4.43 as well as 2.2.34, which were released three and more than six years ago respectively.

While the general methodology described in this paper could still be applied to the older Apache versions, some modifications in our implementation had to be made. First, older versions of Apache utilize `httpd` as their default process name, which was changed to `apache2`

```
$ cat example.com.conf
[...]
            SSLEngine on
            SSLCertificateFile /cert.pem
            SSLCertificateKeyFile /key.pem
    </VirtualHost>
```

```
$ ./extract_server_information.py apache2/dumps
[...]
        error_fname -> /var/log/apache2/error.log
        module_config ->
                [...]
                ap_document_root -> /var/www/example.com
                protocol -> https
        tls_config ->
                ssl_pk_server ->
                        cert_files -> /cert.pem
                        key_files -> /key.pem
[...]
```

Fig. 14. Configuration for a TLS-enabled virtual host.

later. This circumstance has to be considered when validating `process_rec` structures. Secondly, even though the same structures are used in version 2.2.34, the order of their members has changed. For this reason, we have added a heuristic approach to determine automatically, which version of a structure in memory was found. However, even though structures were modified over time, the members and links between them utilized by our methodology have been constant over time.

5. Conclusion and future work

Memory forensics has long been recognized as a valuable tool for investigations, particularly for extracting key information and analyzing malicious processes. Despite the recognition of memory forensics as a desirable component in web server investigations as pointed out by Case et al., in 2017 (Case and Richard III, 2017), little progress has been made in this field since then.

For this reason, this paper explored the potential of memory forensics in the context of web servers by identifying and examining forensically relevant artefacts that can be found in their memory. We focused our analysis on Apache2, one of the most widely used web servers, and developed a unique methodology for extracting crucial information about connections, requests, and configurations from its memory. Our methodology represents the first of its kind and provides a foundation for further exploration and extraction of Apache2 artefacts as well as for analyzing other web server implementations.

We implemented our methodology as a standalone tool, which can be used to analyze dumped process memory. Furthermore, it will be implemented as a Volatility 3 plugin to be able to work on full memory dumps. Additionally, we have introduced a framework for creating various test scenarios to support the development and evaluation of future web server forensic methods. All of these implementations will be released as open-source software and made available on GitHub (Hilgert et al., 2023).

Our evaluation showed that it is in fact possible to extract Apache2 structures from memory, which can be of relevance during an investigation, e.g. when configuration files have been deleted from persistent storage. On the other hand, it also highlighted the limitations of a structured approach, when it comes to Apache2 web server forensics and revealed the potential of unstructured approaches to recover significant remnants such as IP addresses and request lines that persist in memory, even when structured approaches reach their limitations. In the future, it is important to evaluate if and how these traces can be mapped to any possible events that have occurred on the web server.

Furthermore, it is important to note that the results we presented here serve as a lower bound for the artefacts that can be found during an investigation. When a complete physical memory dump is available, our methods would also be able to find certain structures, which have been freed by the Apache process but not yet been reused by the operating system or different processes. The possibilities of an analysis on a full memory dump is subject to further research.

References

Apache Tutor. Introduction to Buckets and Brigades. URL: http://www.apachetutor.org/dev/brigades.

Apache2, 2023. Apache2: main page. URL. https://nightlies.apache.org/httpd/trunk/doxygen/index.html.

Case, A., Richard III, G.G., 2017. Memory forensics: the path forward. Digit. Invest. 20, 23–33.

Hilgert, J.N., Schell, R., Jakobs, C., Lambertz, M., 2023. A framework for web server dataset generation. URL. https://github.com/fkie-cad/apache2-memory-forensics-paper-material.

Kumar, V., Singh, A.P., Rai, A.K., Wairiya, M., 2011. Self alteration detectable image log file for web forensics. Int. J. Comput. Appl. 975, 8887.

Maartmann-Moe, C., Thorkildsen, S.E., Årnes, A., 2009. The persistence of memory: forensic identification and extraction of cryptographic keys. In: Digital Investigation: Proceedings of the Ninth Annual DFRWS Conference, vol. 6, pp. S132–S140.

Nazar, N., Shukla, V.K., Kaur, G., Pandey, N., 2021. Integrating web server log forensics through deep learning. In: 2021 9th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO), IEEE, pp. 1–6.

OpenSSL Foundation, I.. URL. https://www.openssl.org/docs/man1.1.1/man7/ssl.html.

OpenSSL Foundation. URL. Inc.,/docs/man1.1.1/man1/ciphers.html. https://www.openssl.org/docs/man1.0.2/man1/ciphers.html.

Schneider, J., Wolf, J., Freiling, F., 2020. Tampering with digital evidence is hard: the case of main memory images. Forensic Sci. Int.: Digit. Invest. 32, 300924.

Taubmann, B., Frädrich, C., Dusold, D., Reiser, H.P., 2016. Tlskex: harnessing virtual machine introspection for decrypting tls communication. Digit. Invest.: Proceed. Third Annual DFRWS Europe 16, S114–S123.

The Apache Software Foundation. Apache mpm event. URL. httpd.apache.org/docs/2.4/en/mod/event.html.

W3Techs, 2023. Usage statistics and market share of Apache version 2, may 2023. URL. https://w3techs.com/technologies/details/ws-apache/2.