Final Report - Attacking Logging Server from Logging Client

EP284U Ethical Hackning, Project Assignment

Lucas Drufva
KTH Royal Institute of Technology
School of Electrical Engineering and Computer Science (EECS)

November 6, 2024

Introduction

This project [1] describes a comprehensive attack chain that starts with a client computer monitored by the Elastic Stack and culminates in the compromise of confidential data on the Elasticsearch server. By strategically combining two relatively low-severity vulnerabilities, this report demonstrates how their exploitation together poses a significantly greater threat. This project includes a demonstration environment and includes an example attack script designed to exploit vulnerabilities within this setup, providing a practical insight into these vulnerabilities.

Background

Elastic Stack

The Elastic Stack, commonly referred to as the ELK Stack, is an open-source suite of tools developed by Elastic. It is designed to collect, search, analyze, and visualize data from diverse sources in real-time, making it a powerful resource for handling large-scale logging, monitoring, and analytics needs. The stack consists of four main components: Elasticsearch, Logstash, Kibana, and Beats, each playing a specific role within the data pipeline. This modular architecture allows organizations to build a flexible logging and analysis system tailored to various use cases, from infrastructure monitoring to cybersecurity and business intelligence [1]. This report will focus on the Beats and Elasticsearch parts of the elastic stack, as they are directly involved in the exploits, however in a real world scenario they would likely be part of a larger system containing all the parts of the elastic stack.

At the core of the Elastic Stack is `Elasticsearch`, a distributed, RESTful search and analytics engine. Built on top of the Apache Lucene library, Elasticsearch provides robust indexing, querying, and data storage capabilities. It is particularly well-suited for handling large datasets in near real-time, enabling high-performance data retrieval and analysis. Elasticsearch can be scaled horizontally, allowing it to support significant volumes of data by distributing the workload across clusters. It uses a document-oriented approach, where data is stored in JSON format, enabling complex nested data structures and efficient storage of high-dimensional data.

`Beats` are lightweight data shippers that collect and send various types of data from different endpoints to Logstash or Elasticsearch. Each Beat is designed to handle specific types of data, providing modularity and flexibility in data collection. For example:

- `Filebeat` collects and forwards log files.
- `Metricbeat` gathers system and service metrics.
- `Packetbeat` monitors network packets.
- `Winlogbeat` collects Windows Event logs.
- `Heartbeat` performs uptime monitoring by checking the availability of networked services.

Beats are typically deployed on individual servers, containers, or other endpoints, acting as agents that gather data continuously with minimal resource usage. This architecture enables organizations to capture a wide range of operational data with minimal impact on system performance. By sending structured and filtered data directly to Elasticsearch or through Logstash for further processing, Beats play a crucial role in ensuring that data flows efficiently through the Elastic Stack.

Keystore

Elastic Beats use keystores to store credentials and other sensitive information without including them directly in your configuration files. Earlier versions of Elastic's documentation mentioned that "you can use the [beatname] keystore to securely store secrets," [2] leading users to believe the keystore provide strong security guarantees.

However, the keystore values are not securely stored and are instead merely encrypted with a hardcoded default secret [3].

CVE-2021-22145

CVE-2021-22145 is a memory disclosure vulnerability affecting Elasticsearch versions 7.10.0 to 7.13.3 [4]. This vulnerability allows attackers who can submit arbitrary queries to exploit error-handling mechanisms, which may inadvertently expose sensitive data through error messages. Specifically, a malformed query can trigger error messages that reveal previously used data buffers containing potentially sensitive information, such as documents, user authentication credentials, or other private data used in previous requests.

[1] `https://github.com/lucasdrufva/EP284U-project`

Root Cause and Jackson's Role

The origin of the vulnerability traces back to a bug in Jackson, a widely used JSON library in Java and the JVM ecosystem [5]. Jackson's core functionality involves parsing and generating JSON and other data formats, and it provides numerous additional modules for handling various data structures like Avro, BSON, XML, and CSV. Within the Elasticsearch environment, Jackson is leveraged to parse and manage incoming query data.

The issue arises because Jackson's error message generation does not respect buffer offsets and lengths, using the entire buffer content in error messages [6]. Specifically, the `_appendSourceDesc` function in `JsonLocation.java` (version 2.10.4) is at cause. Its responsibility is to build a string for the error message, telling the user the source input string that caused the exception. On line 205, highlighted in yellow in listing 1, a new string is created starting from offset 0 up to the `MAX_CONTENT_SNIPPET` length (500 bytes), disregarding specified constraints [7]. Consequently, this behavior allows for information leakage if an attacker induces Jackson to generate an error.

```
169 protected StringBuilder _appendSourceDesc(StringBuilder sb)
170 {
171     final Object srcRef = _sourceRef;
172      // [Code ommited for brevity]
202      if (srcRef instanceof byte[]) {
203         byte[] b = (byte[]) srcRef;
204         int maxLen = Math.min(b.length, MAX_CONTENT_SNIPPET);
205         _append(sb, new String(b, 0, maxLen, Charset.forName("UTF-8")));
206         len = b.length - maxLen;
207         charStr = " bytes";
208     }
209     // [Code ommited for brevity]
214     return sb;
215 }
```

Listing 1: jackons-core/src/main/java/com/fasterxml/jackson/core/JsonLocation.java

Vulnerability in Elasticsearch's Memory Handling

The risk of data leakage in Elasticsearch is amplified by its use of the Netty framework to manage web traffic. Netty enables efficient network communication by creating shared memory pools for handling HTTP requests and responses. Netty allocates a 1MB data buffer for these operations, implementing custom memory management to avoid overhead from Java's default memory handling. When Elasticsearch uses Jackson to process a request, it passes a reference to the entire Netty buffer, along with an offset and length specifying the relevant subset, to optimize memory usage.

However, when Jackson encounters an error in handling such a request, it generates an error message using the entire buffer's content (from offset 0 up to the specified snippet length of 500 bytes) rather than the specified subset. This behavior could lead to the unintended exposure of sensitive data from previous requests still present in the buffer. Examples of potentially exposed data include logs, query results containing sensitive records, and credentials such as those used in HTTP Basic Authentication.

Patch and Resolution

The Elasticsearch development team addressed this issue in version 7.13.4, released in July 2021, by wrapping the data passed to the Jackson library within a `ByteArrayInputStream` as seen on line 93, listing 2 in the function `createParser` that is responsible for instantiating the Jackson JSON parser[8]. `jsonFactory` in this code refers to an instance of the `JsonFactory` class from Jackson. This approach allows for internal handling of offset and length parameters, ensuring that only the specified subset of data is accessible for Jackson to use, which mitigates the vulnerability from Elasticsearch's side. The pull request adding the change [9] is tagged as a non-issue and with a description having no mention of any intent to solve the memory leak, therefore suggesting the fix may have been accidental.

```
89 @Override
90 public XContentParser createParser(NamedXContentRegistry xContentRegistry
       ,
91         DeprecationHandler deprecationHandler, byte[] data, int offset,
              int length) throws IOException {
92     return new JsonXContentParser(xContentRegistry, deprecationHandler,
93         jsonFactory.createParser(new ByteArrayInputStream(data, offset, length)));
94 }
```

Listing 2: elasticsearch/libs/x-content/src/main/java/org/elasticsearch/common/xcontent/json/JsonXContent.java

By utilizing `ByteArrayInputStream`, this code ensures proper handling of the data bounds and mitigates the potential for unintended data exposure during logging. However, the underlying vulnerability remained within Jackson itself until its full resolution in version 2.13.0, released in September 2021, which included updates to the library's internal buffer data handling for enhanced security during error logging.

Vulnerable Demo Environment

The demo environment developed for this project consists of two virtual machines configured to demonstrate vulnerabilities within a simulated logging infrastructure. Although the environment should be compatible with any modern operating system, the setup has been tested exclusively on Ubuntu 22.04. One machine functions as a log server, while the other serves as a monitored client. This setup provides a controlled environment to study the security issues in logging and monitoring systems, focusing particularly on exploitation related to CVE-2021-22145.

Log Server Configuration

The log server runs `Elasticsearch` version 7.13.3, which is configured to store and manage logs sent from the client machine. Given the exploit use of pooled memory, enough memory must be given to the server otherwise the pooling feature is disabled. This server has been verified working with a minimum of 4 GB of RAM.

Two users are configured on the log server:

- `elastic`: an administrator with full privileges, allowing complete access to manage and configure the Elasticsearch service.
- `filebeat_user`: a lower-privilege user designated solely for log ingestion purposes, with limited permissions appropriate for data uploading and indexing.

Monitored Client Configuration

The monitored client machine runs the latest compatible stable version of `Filebeat` (being 7.17.25 at the time of writing), configured to collect and forward system logs to the log server. `Filebeat` authenticates to Elasticsearch using native basic authentication with the credentials stored securely in a keystore. This setup obfuscates the password and limits exposure in configuration files.

The client uses the `filebeat_user` account, which has only the necessary privileges for log shipping. To compromise this configuration, an attacker would need root access to the client machine or, at a minimum, read permissions on the `Filebeat` configuration files. In this project setup, root access is granted on the client to simplify testing and demonstration of the vulnerability.

Simulated Exploit of CVE-2021-22145 Vulnerability

The vulnerability targeted in this setup, `CVE-2021-22145`, allows a limited 500-byte memory leak from Elasticsearch. This exploit is simulated by injecting sensitive data into the memory space of Elasticsearch. A script on the log server machine periodically sends HTTP requests to the Elasticsearch server, authenticating as the administrator user (`elastic`), thus introducing sensitive data into memory.

Since `CVE-2021-22145` only leaks the first 500 bytes of a 1MB buffer, specific conditions must be met to ensure the sensitive data falls within the leaked portion of memory. Two methods were considered for achieving this:

1. Ensuring that the HTTP request containing sensitive data is the first one processed by Elasticsearch after startup, thus placing it at the start of the buffer.
2. Sending a large number of requests to fill the buffer and reset it, placing the sensitive data in the beginning of the memory space.

For simplicity, the first method was adopted. However, this approach introduces a race condition where the `Filebeat` logs from the client might reach the server before the sensitive HTTP request, thus disrupting the memory placement. To mitigate this, a looped process was employed: Elasticsearch starts, the sensitive request is sent, a delay occurs, and Elasticsearch shuts down, restarting the process. In a real-world scenario, the exploit would be run repeatedly until sensitive data randomly appears in the exposed memory area, potentially aided by filling the buffer with benign data.

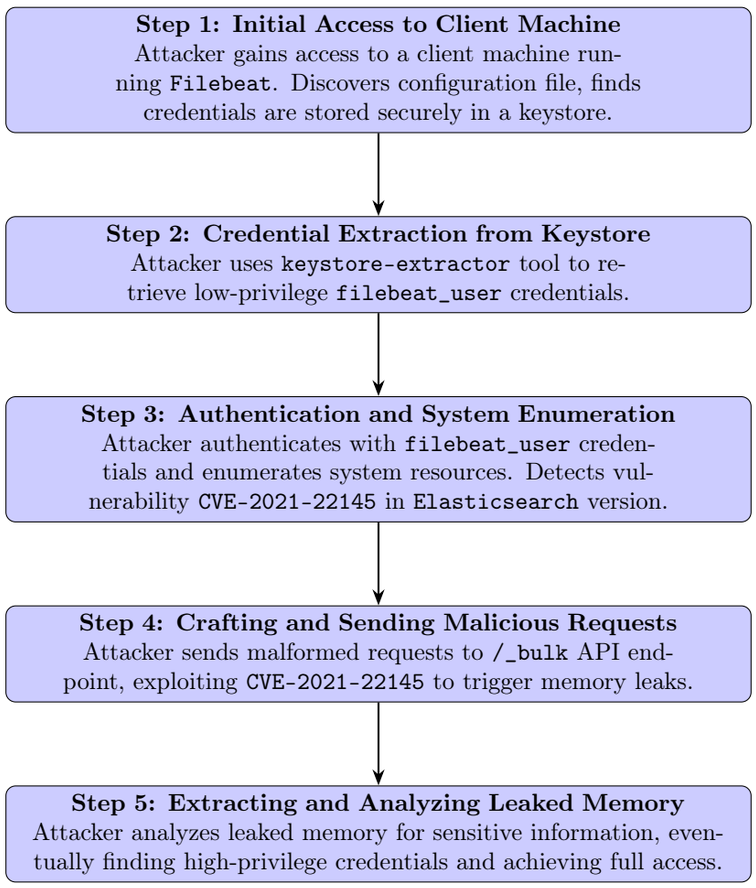
Network Configuration and Access Control

The network setup restricts access so that only clients inside the network can communicate with the log server, and solely on port 9200, which is the default port for Elasticsearch. This limitation is intended to reflect a realistic security scenario, where direct access to Elasticsearch servers is generally restricted. Such a configuration also emphasizes that the challenge lies in exploiting Elasticsearch to gain unauthorized access to the log server.

In the context of utilizing this demo as a Capture the Flag (CTF) challenge, the network constraints reinforce the focus on breaching the Elasticsearch service as the primary pathway to accessing sensitive log data on the server. This design isolates the log server, mirroring real-world environments where Elasticsearch servers are typically accessible only within trusted networks, preventing direct external access.

This demo environment, thus, serves as an effective simulation platform for demonstrating security weaknesses in logging architectures, highlighting specific techniques to exploit buffer-based vulnerabilities under controlled conditions.

High-Level Attack Flow



Assumptions and Delimitations

- **Focus on Elastic Stack Components**: This project is limited to exploring vulnerabilities within components of the `Elastic Stack`, specifically `Filebeat` and `Elasticsearch`. Other potential attack surfaces or components in the environment are excluded to maintain focus on these specific elements.
- **Access to Filebeat Configuration Files**: It is assumed that the attacker has direct access to `Filebeat` configuration files on the client machine, allowing them to proceed with the exploit without the need for initial privilege escalation. In a broader scenario, attackers might first need to escalate privileges on the client machine to access these configuration files.
- **Simplified Authentication Method**: The demo environment uses HTTP Basic Authentication for simplicity. While this project demonstrates an attack using basic authentication, similar principles could apply to other authentication methods, such as token-based or certificate-based mechanisms, with appropriate modifications.

- **Predictable Memory Placement of Sensitive Data**: The project assumes that sensitive data, such as credentials, are relatively accessible within the memory leaks triggered by the vulnerability. In a real-world scenario, the placement of sensitive data in memory would be less predictable, and the attacker would likely need to send a significantly higher volume of requests to increase the likelihood of capturing useful information in leaked memory.

Demo Execution Process

To demonstrate the attack in practice, an automated environment setup and exploit process has been created. This automation is managed via `Vagrant` and `Ansible`, allowing seamless deployment of both the vulnerable environment and the exploit process. Below is a detailed description of the demo execution steps.

Environment Setup

The demo environment is initiated by running the command `vagrant up`. This command provisions and configures both the log server and the client machine, establishing the necessary network and user configurations for the attack simulation. The setup can take approximately 5 minutes to complete, depending on system resources and network conditions.

Launching the Exploit

The exploit is launched with the command `vagrant exploit`. This command initiates an automated exploit process using `Ansible` over `SSH` on the client machine. Since the exploit relies on timing and memory manipulation, this process can take up to 15 minutes to run fully. The automation framework ensures each step is executed in the correct sequence with no user intervention.

Extracting Keystore Credentials

The exploit begins by cloning the `lucasdrufva/keystore-extractor` repository from GitHub on the client machine. This tool is built using `Golang` and is necessary to extract stored credentials from the `Filebeat` keystore file, `/var/lib/filebeat/filebeat.keystore`. Once built, the `keystore-extractor` is executed as seen in listing 3, and the resulting output is parsed to obtain the credentials for the `filebeat_user` account, which has limited access to the Elasticsearch server.

```
$ ./keystore-extractor /var/lib/filebeat/filebeat.keystore
ES_PWD : oTCLMRVPO6
```

Listing 3: Execution of keystore-extractor tool

Preparing and Sending the Exploit Payload

After obtaining the `filebeat_user` credentials, the exploit generates an HTTP Basic Authentication header. The credentials are then used to authenticate to the Elasticsearch server. To trigger the vulnerability `CVE-2021-22145`, the exploit crafts a payload designed to trigger a memory leak via the Jackson library's error handling.

The payload consists of malformed JSON, followed by a newline character. This structure causes an error in Jackson's parsing logic when sent to the `/_bulk` API endpoint on the Elasticsearch server, which accepts requests from the low-privilege `filebeat_user`. The response contains an error message, where the `reason` field reveals leaked memory data due to improper buffer handling. See listing 4 for an example response containing a error message that leaks data.

```
{
   "error":{
      "root_cause":[
         {
            "type":"json_parse_exception",
            "reason":"Unrecognized token \\'exploit\\': was expecting
               (JSON String, Number, Array, Object or token \\'null\\'
               , \\'true\\' or \\'false\\')\n at [Source: (byte[])\"
               HTTP/1.1 200 OK\r\ncontent-type: application/json;
               charset=UTF-8\r\ncontent-length: 545\r\n\r\non: Basic
               ZWxhc3RpYzp2S21uZXVOMWZG\r\n\r\n\\x00\\x00\"[truncated
               1048076 bytes]; line: 1, column: 6]"
         }
      ],
      "type":"json_parse_exception",
```

```
      "reason":"Unrecognized token \\'exploit\\': was expecting (JSON
         String, Number, Array, Object or token \\'null\\', \\'true\\'
          or \\'false\\')\n at [Source: (byte[])\"HTTP/1.1 200 OK\r\
         ncontent-type: application/json; charset=UTF-8\r\ncontent-
         length: 545\r\n\r\non: Basic ZWxhc3RpYzp2S21uZXVOMWZG\r\n\r\n
         \\x00\\x00\"[truncated 1048076 bytes]; line: 1, column: 6]"
   },
   "status":400
}
```

Listing 4: Response from Elasticsearch containing leaked data

Parsing and Repeating Payload Requests

The memory leak generated in the `reason` field of the error response is parsed by the exploit script, which scans the leaked data for any sensitive information, particularly HTTP Basic Authentication headers. If a valid header is found, the exploit decodes it, revealing the high-privilege credentials needed to access sensitive data on the Elasticsearch server. In listing 4 a basic auth token is leaked two times.

To increase the likelihood of obtaining relevant data in the memory leak, the exploit sends multiple payload requests over time, with intervals in between. This repetition ensures that eventually, a leaked memory segment may contain sensitive credentials, demonstrating the feasibility of exploiting `CVE-2021-22145` to access restricted data.

Summary

This automated exploit process illustrates the vulnerability's real-world implications, showing how unauthorized access can be gained through systematic memory leakage and credential harvesting. The demo environment provides a practical example of the attack flow, demonstrating how timing and payload manipulation enable attackers to reveal sensitive data on a vulnerable Elasticsearch server.

References

[1] Dotan Horovits. *The Complete Guide to the ELK Stack*. URL: https://logz.io/learn/complete-guide-elk-stack (visited on 10/21/2024).

[2] cmacknz. *Clarify that the keystore obfuscates secrets only*. 2024. URL: https://github.com/elastic/beats/pull/38667 (visited on 10/21/2024).

[3] Elastic. *elastic-agent-libs/keystore/file_keystore.go*. 2022. URL: https://github.com/elastic/elastic-agent-libs/blob/be95ccab81f44833cf150962da3003394c13c6ea/keystore/file_keystore.go#L98 (visited on 10/21/2024).

[4] National Institute of Standards and Technology. *CVE-2021-22145 Detail*. 2021. URL: https://nvd.nist.gov/vuln/detail/CVE-2021-22145 (visited on 10/22/2024).

[5] FasterXML. *Jackson*. 2024. URL: https://github.com/FasterXML/jackson (visited on 10/29/2024).

[6] Greg Wittel. *Misleading exception for input source when processing byte buffer with start offset (see JsonProcessingException)*. 2020. URL: https://github.com/FasterXML/jackson-core/issues/652 (visited on 10/29/2024).

[7] FasterXML. *jackson-core/src/main/java/com/fasterxml/jackson/core/JsonLocation.java*. 2017. URL: https://github.com/FasterXML/jackson-core/blob/jackson-core-2.10.4/src/main/java/com/fasterxml/jackson/core/JsonLocation.java (visited on 10/29/2024).

[8] Elastic. *elasticsearch/libs/x-content/src/main/java/org/elasticsearch/common/xcontent/json/JsonXContent.java*. 2021. URL: https://github.com/elastic/elasticsearch/blob/v7.13.4/libs/x-content/src/main/java/org/elasticsearch/common/xcontent/json/JsonXContent.java (visited on 10/29/2024).

[9] Armin Braun. *Dry Up XContent Parser Construction*. 2021. URL: https://github.com/elastic/elasticsearch/pull/75114 (visited on 11/01/2024).