# ATTACKING THE MESSENGER: EXPLORING THE SECURITY OF BIG DATA MESSENGER APACHE KAFKA

A Thesis
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Tyler Losinski

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Program:
Software Engineering
Option:
Cybersecurity

September 2021

Fargo, North Dakota

# North Dakota State University
Graduate School

**Title**

ATTACKING THE MESSENGER: EXPLORING THE SECURITY OF BIG DATA MESSENGER APACHE KAFKA

**By**

Tyler Losinski

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

**MASTER OF SCIENCE**

SUPERVISORY COMMITTEE:

Jeremy Straub
Chair

Kendall Nygard

Daniel Klenow

Approved:

12/23/2021
Date

Simone Ludwig
Department Chair

## ABSTRACT

As technology becomes faster, cheaper, and more compact, a higher volume of data must be processed. This demand drives the need to process high volumes of data in near real time. As a result, technologies such as Kafka have been created as high throughput messaging bus systems. Utilizing these new technologies could vastly improve the way we look at data processing, especially when that data is coming from IoT or distributed systems.

Kafka provides multiple methods of encryption and authentication with its brokers, however, securing the producers and consumers is the responsibility of the application owner. This paper focuses on this key aspect in order to consider how an attacker could exploit and compromise the flow and integrity of the data. After access to the producers and consumers has been compromised, examples of data manipulation are performed in order to demonstrate real world consequence of breaches such as these.

## ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Jeremy Straub who helped me through the process of writing this paper. His help and guidance has helped me immensely in my graduate program and I couldn't have completed all that I have without him. I also appreciate all the support and help I received from Bobcat as they helped me financially throughout my thesis and graduate program.

## TABLE OF CONTENTS

## LIST OF FIGURES

# 1. INTRODUCTION

The size of the internet is estimated to be 16 million petabytes of information [1]. It is estimated to be growing at a rate of 70 terabytes per second [1]. These numbers show an explosive growth never before seen in the internet's history. The creators of LinkedIn saw this challenge and developed a high throughput messaging bus system called Kafka, to handle the processing of log files. After realizing the need for and importance of what they created they eventually released this technology under the Apache license to allow anyone to utilize it. Today more than 12,000 companies use Kafka to stream and process their data in real time [2].

One area that could benefit greatly from this system is the capture of data from various sensors. Manufacturing systems, for example, have thousands of sensors with functions ranging from collecting temperatures, checking fuel levels, to even tracking when people enter or exit a building [3]. All this data can be collected and sent to one Kafka cluster where it can be consumed by various applications, and either be processed as it comes in or stored to be processed later for use in a report or used by another system. Having the ability to capture all this data and the freedom to process it when and where the user wants makes Kafka a top contender in this data driven world.

Kafka excels in its high throughput capabilities and its distributed nature also makes it perfect for scenarios where fault tolerance is key. While collecting and processing high volumes of data is great, being able to recover if a computer fails can be even more important. That is why Kafka runs in a clustered environment across one or more servers. This not only allows the cluster to continue to run the case of a fault, but it also allows the distribution of consumers and producers across many systems and instances. For example, a system might collect messages from a thermostat every second then consume and enrich that data to be used by another process.

That consumer can be replicated across many servers to prevent the failure of one server from bringing down the entire enrichment process. Furthermore, this process can be utilized to scale the system if for some reason the thermostat started producing more data during a specific time of day than normal. This scalability coupled with its enormous throughput makes Kafka idea for modern data driven problems.

This paper takes a deeper look into the data processing and distributed nature of Kafka and compares it with more common and traditional data processing techniques. After this, potential security vulnerabilities are explored. Traditionally, data is collected and stored in a structured or unstructured database where a user must query the database to get the data that is required. This can be a costly endeavor as the database usually can't push data to the user unless it is explicitly requested. Another issue is that every time the data is requested the query needs to be reprocessed which, depending on the efficiency of the query, can waste valuable time and resources. However, if the data is processed as it is received, the need for a centralized data processing system is reduced.

The proposed system utilizes Kafka to process data as it is received. This will be achieved by consumer/producer services that perform simple calculations such as counting, data enriching data, and data filtering. After this, the data is pushed to Kafka topics to either be pulled for reporting or reprocessed and pushed to another topic.

## 2. BACKGROUND AND RELATED LITERATURE

Although there has been an explosion of data growth over the last few decades [4], securing that data has not been at the forefront of that explosion. According to an article published in the International Journal of Emerging Trends in Social Sciences, in 2019 there were roughly 1244 data breaches across multiple industries which resulted in the compromise of 446,515,334 records [4]. This has led to an estimated average cost of $3.9 billion in losses with the highest single incident loss being $8.19 billion [5]. With stakes as high as these, it is imperative that securing data is put at the forefront of research [6].

Much of this generated data has moved to IoT devices as technology becomes ever more connected [7]. With this, there is a need to for technology to manage huge volumes of data with a high level of fault tolerance. This is the drive behind why Kafka was developed.

### 2.1. What is Kafka?

Kafka has been utilized by thousands of companies to bring real time data processing to massive concurrent data streams [8]. This relatively new technology was developed by LinkedIn as a new approach to handling their log files. As a result, Kafka leverages many new methodologies and features to create a system with high throughput and fault tolerance. Through its innovative producer consumer architecture, Kafka can leverage multiple servers and services to create a system that can manage large streams of data [9].

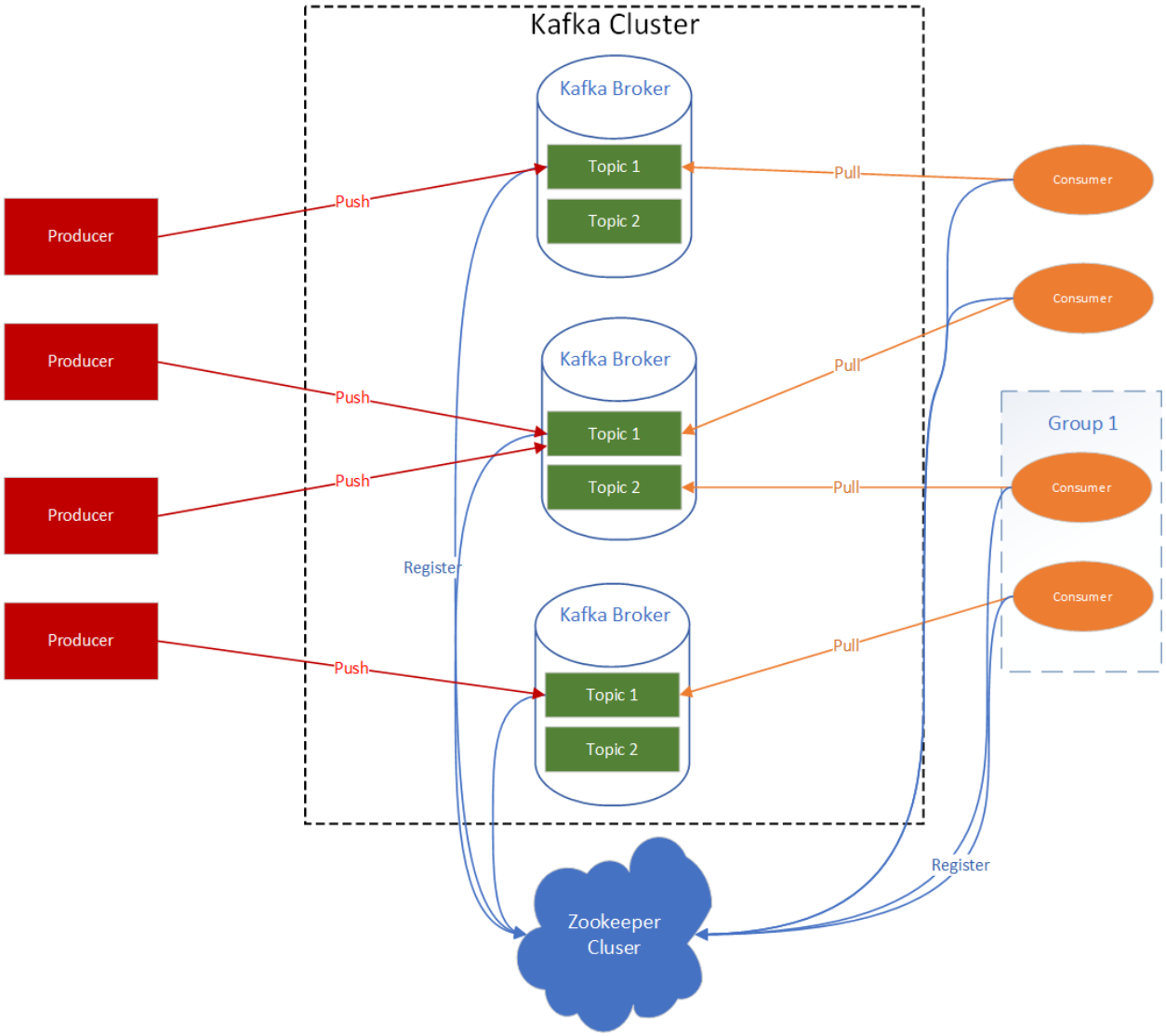


Figure 1. Kafka Architecture Example

Depending on the demand, Kafka's brokers can be scaled to allow for more throughput and data redundancy. This can be done in order to accommodate more data streams and/or provide fault tolerance. This scaling can also be performed on the consumers and producers of the system to achieve the same effect [10].

As can be seen in Figure 1, topics can have one or many partitions which directly correlate to their throughput. They can also have a replication factor of one or more in order to

ensure there is redundancy between the brokers. Consumers and producers utilize these topics to read and write data. These consumers and producers can be run in one or many locations, similar to the topic's architecture, to ensure redundancy and throughput. The consumers and producers can be written in many different programming languages and can be run across many different systems. This diversity can leave the system vulnerable to attacks.

A Kafka service starts the process by publishing raw data to a topic. Once that data is available, Kafka will push the data to any consumers listening to that topic. At this point, another Kafka services will consume the raw data and can perform the necessary data manipulation such as counting specific records, enriching the data from a database, or filtering out data that is unnecessary. After this data manipulation, the Kafka service will publish the new data to a different topic where the process can begin once again. Another use a consumer might serve would be to push the data in real time to a reporting application, which is where a system such as this would excel [11].

This process has the potential to process data before the user requests it, allowing for near real time fetching of processed data. Comparing this to a system where data manipulation and joining needs to occur every time a report is generated, there is the potential for huge time savings [12]. Even using this process to make complex calculations then joining that data with a dataset from a relational database could provide time savings. Not only does this system have the potential to save time, but it also has the potential to offload processing to edge services.

Given the distributed nature of Kafka consumers and producers, it is possible to move these services to be either client side or on lightweight edge services. The benefits of a system like this include the ability to scale up or down processing nodes based on demand. This alone could save an enterprise system thousands of dollars a year [13]. Along with the cost savings,

this system could utilize technology, such as IoT devices, that could process data while they are not in use.

In contrast, traditional relational databases lack the ability to offload processing to smaller services, giving traditional relational data storage a distinct disadvantage [14]. Managing one single point of processing reduces the overall complexity of the system but does reduce its ability to expand on demand. Also, database engines can quickly generate datasets that have been filtered and aggregated on demand whereas, distributed systems such as Kafka would require complex systems of consumers and producers just to generate one set of data. Overall, both systems have their own specific pros and cons, and these should be carefully considered when deciding which to implement.

Although Kafka is compared directly with relational databases in this paper, there is the potential to utilize more recent technologies that use No-SQL architectures. Technologies such as MongoDB are known for their speed at scale [15] which might work well with Kafka. However, this topic will be left to future work as it adds a level of complexity and exceeds the scope of this paper.

## 3. PROPOSED SYSTEMS

This paper explores three different areas of Kafka and considers its potential benefit over traditional data storage and the potential security risks it brings. With security misconfiguration often cited as one of the top security risks, the following example systems have a resemblance to many current production systems [16]. OWASP cites these types of vulnerabilities as having the highest likelihood of being prevalent and detectable. This unfortunate truth further underlines the importance of securing these types of systems.

### 3.1. System Compared to Traditional Data Storage

The proposed system utilizes a Kafka cluster with three servers running CentOS 7. Each runs a single instance of Zookeeper and 3 Kafka nodes. Each topic will be running with a replication factor of one and will contain nine partitions. This system is compared with a Microsoft SQL Server 2016 Standard Edition server.

The relational database contains two tables, one with the hard drives' information and another that is used to store each day's data and whether it failed or not. The Kafka cluster uses two topics for storing data. The first is used to store the whole year of data and the second is used to store the current count of unique hard drives that have not failed.

The hard drive statistics are from 2018 hard drive statistics data [17]. This dataset contains roughly 100,000 hard drive statistics per day. Each record contains about 53 columns, however, for these tests we will only be using the first five columns which are the data, serial number, model number, capacity (in bytes), and failure status of each hard drive. The failure column is zero if it did not fail that day and one if a failure occurred. Once a failure has happened, the hard drive is removed and will not be seen in subsequent days.

A C# .Net application is used to bulk insert the hard drive info into the SQL server database. This requires around ten to fifteen seconds of processing per 100,000 records. Another bulk insert was used to import the hard drive status per day and whether a failure had occurred, which was much faster due to not having to check for unique records. This tool takes around five to ten seconds. After the data was in the SQL Server database, the C# application was once again used to push each record to a Kafka topic at a rate of about 100,000 records per one to five seconds.

To process the data in SQL server, a simple count statement was written to get all the hard drives that had not yet failed. This was then run through a simple C# application and the time to retrieve the data was recorded.

To process the data in Kafka, a Node.js service was written to consume all the messages that had been sent to the topic containing the raw data. The services then created an object of hard drives, adding new hard drives to it as they came in and removing drives that had failed. For each record that came in, another Kafka record was produced to a counts topic. This was a simple record containing the date of the hard drive stats that were consumed and the current count of failures. This produced a timeline of counts each time a new record is available to process. Overall, the process used to transform and filter the data, produces a simple count on active drives; however, this is a complex process. If the complexity of the system could be overcome, the time saving benefits could be great.

### 3.2. System for Exploiting Producers & Consumers

In order to demonstrate a denial of service (DoS) attack, a consumer was run on a CentOS 7 server with minimum system requirements. The server had 512 MB of RAM, 10gb of hard drive space, and one CPU core (an Intel Xeon E5-2407 at 2.20Ghz). This system represents

a consumer that may only perform a simple yet important task such as sending out an email regarding critical errors or storing data in a persistent database. This paper presents a consumer written in Java that continually writes its received data to a text file.

The server running the Kafka message bus also ran CentOS 7 but with 16GB of ram, 126 GB of hard drive space, and 4 CPU cores (Intel Xeon E5-2407 at 2.20Ghz). Both the message bus and consumer were run through Hyper-V on the same Windows Server 2016 machine. The producer of the messages was run on the same server as the message bus using the included Kafka Producer script. The consumer used a Node.js Kafka package called kafkajs [18].

### 3.3. System for Target Malware Attacks

To demonstrate a potential system that can be exploited, a Kafka cluster was running on a single virtualized server running CentOS 7. The cluster only ran one instance of Zookeeper with one node. The resources available to the server were 16Gb of RAM and 4 CPU cores (Intel Xeon E5-2407 at 2.20Ghz). For the purpose of these experiments, several ‘dummy’ topics were defined with various partition and replication sizes. These topics represented a typical system that is consuming and managing real world data.

The software that represents potential malware was a custom written C# application utilizing .NET 4.5. This application represents what potential malware could do to attack an unsecured Kafka cluster on a network. Such an application would have a means to spread to other systems and automatically run these steps in order to discover and exploit any Kafka systems on the network. Furthermore, this application design assumes that the message bus is unsecured; however, to maximize effectiveness, leveraging a vulnerability to exploit Kafka cluster and connect without authorization would greatly increase its effectiveness. This is left to future research.

This type of malware attack is considered spyware, as its main task is to enumerate and send information back to the attacker. Along with gathering reconnaissance information, it relies heavily on a default configuration attack of the Kafka systems, which is cited in the OWASP top 10 vulnerabilities year after year [16]. The combination of these common cybersecurity issues makes this type of malware plausible.

# 4. EVALUATION

This section covers three major points of evaluation. First, a comparison is made with traditional means of data storage. Next, an evaluation of attacking weak consumers is explored. Finally, examples of malicious code are examined to show how attacks can be automated.

## 4.1. Traditional Database Comparison

When evaluating the proposed system, it became evident quickly that complexity would pose a challenge in testing. The two components of the tests are the Kafka cluster, with its consumers and producers, and a SQL Server database. Using these two systems, a comparison was made as to how quickly each could count the number of failures over the course of a year and how many operating hard drives currently exist. Although this seems like a trivial test, Kafka's immutable nature of the data within a topic resulted in complicated consumers and producers while SQL Server handled these tests with ease.

With the SQL Server, the hard drive failure data was mass loaded into two tables. The first table, named HardDrive, held all the unique hard drives the data set contained. The second table, HardDriveStatus, held the status of each hard drive on a given day. The following query was used to get a count of failures of the year and its elapsed time.

```
  BEGIN
       DECLARE @t1 DATETIME;
       DECLARE @t2 DATETIME;

       SET @t1 = GETDATE();

       SELECT COUNT(*)
       FROM HardDriveStatus
       WHERE FAILURE = 1;

       SET @t2 = GETDATE();

       SELECT DATEDIFF(millisecond,@t1,@t2) AS elapsed_ms;
END;
```

The results from this query were 117,993 records and a time of 5517 milliseconds. These results show just how simple it is to query this data from a traditional structured database.

```
BEGIN
	DECLARE @t1 DATETIME;
	DECLARE @t2 DATETIME;

	SET @t1 = GETDATE();

	SELECT COUNT(*) FROM (
		SELECT HDS.SERIALNUMBER FROM HardDriveStatus HDS
		JOIN HARDDRIVE HD
		ON HDS.SerialNumber = HD.SerialNumber
		WHERE Failure != 1
		GROUP BY HDS.SerialNumber
	) X

	SET @t2 = GETDATE();

	SELECT DATEDIFF(millisecond,@t1,@t2) AS elapsed_ms;
END;
```

Kafka tests were not as straight forward or simple to create. First, all of the data was pushed into a topic called Hard_Drive_Data via a simple producer written in Node.js. Then a consumer was written that would pull in each record from the Hard_Drive_Data topic, count the unique hard drives that were still operational and push that information into a topic called HardDrive_Count. This was accomplished by holding a temporary object of hard drives that would be added to or removed from each time a hard drive was added or failed. Due to its complexity and limited resources, no conclusive results were recorded.

A similar process was followed when counting the number of failures over time by consuming in the raw data from the Hard_Drive_Data topic, counting the number of failures and pushing them back to another topic. However, no conclusive results were gathered from these tests due to the complexity and limited resources constraint seen in the previous results.

Unfortunately, due to the complex nature of data streams and the required resources to manage such large datasets, no conclusive data can be drawn from our simple Kafka test. As outlined in the future work section, a better solution might include a mixture of each experiment

where a SQL database generates the initial data, and a Kafka topic supplements the subsequent data. However, this type of experiment helps outlines the distinct differences Kafka and SQL Databases have in architecture. This further emphasizes the need to continue to develop the research and understand around how Kafka differs in vulnerabilities and exploits as compared to more traditional architectures.

### 4.2. Attacking Weak Consumers

A simple consumer was constructed to read data from the topic “test_topic” which included 21 partitions and a replication factor of one for high throughput. Then, a consumer was written using the kafkajs package in order to listen in on this topic and perform a simple write to file if the messages contained the word “error” in them. This was used to represent a simple service that logs critical system errors for future evaluation and could be improved to also send out an alert via email to key actors. However, for the sake of this experiment it remained simple and only wrote the errors to file. Figure 2 is the code snippet that was used to execute this task:

```
const run = async () => {
  await consumer.connect()
  await consumer.subscribe({ topic, fromBeginning: true })
  await consumer.run({
    eachMessage: async ({ topic, partition, message }) => {

      if(message.value.includes(`error`)){
        fs.appendFile('message.txt', message.value + "\n", function (err) {
          if (err) throw err;
        });
      }
    },
  })
}
```

Figure 2. Example of Error Logging to Kafka Written in Javascript

After the consumer was written it was placed onto the CentOS 7 system with minimal system resources. A file was then loaded into the topic that contained 479,000 words [19] which of those 479,000 words only 79 of them contained the word error; thus, only 79 words were written to the file. This demonstrates a typical system load where critical errors occur infrequently.

This setup poses some vulnerabilities to DoS attacks. To demonstrate this, the word file was replaced with a file that contains 500,000 entries of the word “error”. This file was loaded into the same topic as before, however the consumer cannot handle the volume and inevitably crashes.

Kafka can process huge volumes of data in real time allowing for faster response times and an overall better user experience. However, this distinct advantage could turn into a critical flaw if the code used to consume, manipulate, and produce the data isn’t effectively written [20].

As demonstrated, DoS attacks can pose a serious risk to smaller, more vulnerable services in the system. This risk could then cascade into more detrimental problems such as data not being adequately processed or errors spiraling out of control bringing down more critical systems.

These issues can easily be mitigated by implementing simple standards and/or policies. For the given example, a more robust service could be written that can recognize an overloaded topic and notify someone before the volume causes a crash. A consumer could be written to simply count the throughput of topics and send out notifications when an abnormal amount of traffic is detected. Another approach would be to monitor any errors that the individual services produces, notifying a user when errors start to pile up. Finally, the best approach would be to write the consumers in a way that allows them to have a throughput that is much higher than its

anticipated usage. This might be in the form of synchronous file writes or dropping messages to maintain integrity of the consumer as a whole.

### 4.3. Analyzing the Malicious Code

As stated in the previous section, this code is merely a representation of techniques that could be utilized to maximize the effectiveness of automated attacks. This automation can be broken down into three functions. The first is a scan of the network's local subnet. This could be expanded to include several ranges of IP address or even a specific range if more knowledge is known about the target. Figure 3 is an example of such code:

```
string ipBase = "192.168.0.";
for (int i = 1; i < 255; i++)
{
      string ip = ipBase + i.ToString();

      Ping p = new Ping();
      p.PingCompleted += new PingCompletedEventHandler(p_PingCompleted);
      countdown.AddCount();
      p.SendAsync(ip, 100, ip);
}
countdown.Signal();
countdown.Wait();
```

Figure 3. Code Example of Scanning Network for Potential Kafka Server. Written in C#

This sample of code takes in a base IP address range to scan and checks to see if there is a device connected to each IP address in the range. If a device is discovered, then the following code checks for any open ports:

```
/// Check for open Kafka Port
/// </summary>
/// <param name="ip"></param>
private static void PortScan(string ip)
{
    TcpClient Scan = new TcpClient();
    int[] Ports = new int[]
    {
        9092
    };
    foreach (int s in Ports)
    {
        try
        {
            Scan.Connect(ip, s);
            Console.WriteLine($"[{s}] | OPEN");
            GetKafkaTopicsAsync(ip);
        }
        catch
        {
            Console.WriteLine($"[{s}] | CLOSED");
        }
    }
}
```

Figure 4. Code Example to Scan Open Ports. Written in C#

The only port scanned was 9092, in this example, but this could be expanded to any array of ports. If this scan discovers any open ports it then runs the code in Figure 5 to see if a connection to Kafka can be established:

```
private static async void GetKafkaTopicsAsync(string ip)
{
    var conf = new ConsumerConfig
    {
        BootstrapServers = ip+":9092",
    };
    var c = new AdminClientConfig(conf);

    using (var adminClient = new AdminClientBuilder(
        new AdminClientConfig { BootstrapServers = ip + ":9092" }).Build())
    {
        var meta = adminClient.GetMetadata(TimeSpan.FromSeconds(20));
        Console.WriteLine($"{meta.OriginatingBrokerId} {meta.OriginatingBrokerName}")
;
        meta.Brokers.ForEach(broker =>
            Console.WriteLine($"Broker:
                {broker.BrokerId} {broker.Host}:{broker.Port}"));

        meta.Topics.ForEach(topic =>
        {
            Console.WriteLine($"Topic:
                {topic.Topic} {topic.Error}");
            topic.Partitions.ForEach(partition =>
            {
                Console.WriteLine($"Partition:
                    {partition.PartitionId}");
                Console.WriteLine($"Replicas:
                    {ToString(partition.Replicas)}");
                Console.WriteLine($"InSyncReplicas:
                    {ToString(partition.InSyncReplicas)}");
            });
        });
    }
}
```

Figure 5. Code Example to Check for Available Topics on Discovers Kafka Server. Written in C#

This is where the bulk of the code resides. This method takes in the IP address that we discovered to have the port 9092 open on and attempts to connect to it using the package Confluent.Kafka [21]. If this is successful, the application then attempts to capture all of the available brokers in the system and logs them to the console. At this point, the application could

open a connection with a database or generate some message to send back to the attacker with this information, however, for demonstration purposes all of this data will be logged to the console. Once the brokers are logged, the next step is to iterate through the available topics. Each topic has data such as topic name, partitions, and replicas that are all valuable to know. Once each of these are logged the application exits.

This application shows how easily it is to enumerate and collect data from a network automatically. Although, as stated, this type of attack is only effective against systems that use the default ports and configurations of Kafka. Given that security misconfiguration is one of the top ten OWASP security risks [16], this assumption can still be quite realistic.

After running this application on the test network, the server running Kafka was discovered and successfully enumerated. The application identified four topics, “test-topic, test_topic, topic-test, and topic_test” and their associated partitions. This whole process took 21,047 milliseconds and found 14 active devices on the network. To reduce runtime, limiting the number of hosts scanned and or ports scanned could improve efficiency, at the cost of missing potential targets. However, 21 seconds is an acceptable time if the malware is effectively hidden on an infected target.

Expanding the range of IP addresses could broaden the search, thus improving the chances of finding a Kafka Cluster on a network. However, it may be a better use of resources to focus on the subnet the computer is on and assume the software will propagate across machines throughout the network. This would help reduce suspicious activity on the network and allow the malware to extend its reach faster.

Although this application only enumerates topics and sends that info to the attacker, this setup could be expanded to connect to the discovered topics and send their data to the attacker.

This could be achieved by checking the size of the topics and only siphoning the data if the amount of data is manageable. However, if the attacker creates their own Kafka server specifically to consume all the data from host, all data could be stolen in real time with little to no indication of a breach. This type of attack would require future research to understand its effectiveness.

## 5. CONCLUSIONS AND FUTURE WORK

There is potential for a Kafka based system to have distinct benefits over a traditional reporting system. However, this research shows that the complexity of the solution detracts from its usefulness. This complexity coupled with inconclusive results leads to a conclusion that Kafka alone should not be used as a platform for reporting. There needs to be way to persist data that allows for easy data possessing at the time of the request. With this concept an alternative system could be achieved with less complexity and real time data manipulation.

There is a potential of exploiting Kafka's producers and consumers to cause a catastrophic compromise of the system. Based on the basic experiments conducted, it is easy to draw the conclusion that Kafka doesn't need to be exploited directly to cause failure. Going after the consumers and producers that make use of the Kafka message bus provides far more success when trying to attack a Kafka cluster. Leveraging Kafka's high throughput to perform a successful DoS attack puts the pressure on the developers of the consumers and producers to write them in such a way as to expect overwhelming levels of messages, even when this is not their typical behavior.

As illustrated using a basic C# application, the process of enumerating a network for potential targets can be performed relatively quickly. Once a target is identified, the process of enumeration and data siphoning is a trivial task that could have devastating consequences. However, substantial research still needs to be performed in order to make this type of attack more feasible. A better means of attacking secured messaging busses would be required in order to effectively attack most Kafka clusters. Nevertheless, targeted malware is a type of automated attack that should be considered when trying to secure any system.

Although system misconfigurations were identified as possible vulnerabilities, Confluent has written an extensive catalog of best practices that address these [22]. In a more recent article, they address many of these vulnerabilities by simply implementing authentication [22]. Although these resources are still being built, a simple Google search for the best practices could help mitigate many attacks. Doing an assessment of security in the same manner as how SCADA systems are evaluated could also help to identify and mitigate risks [23].

Kafka can pose numerous benefits over traditional databases, but it does not come without risks. This paper illustrates that but there is plenty of work to be done to leverage Kafka to its most secure and efficient potential. Exploring a combination of traditional storage and Kafka, along with securing Kafka properly, could create robust, fault tolerant, real time data storage systems that could be scaled simply by adding more servers. However, as demonstrated, if these systems are not secured properly, efficient, and even automated attacks can expose this sensitive data.

## REFERENCES


[1] "How big is the internet? — Starry Blog," [Online]. Available: https://starry.com/blog/inside-the-internet/how-big-is-the-internet.

[2] "Companies using Apache Kafka and its marketshare," [Online]. Available: https://enlyft.com/tech/products/apache-kafka.

[3] M. Syafrudin, G. Alfian, N. L. Fitriyani and J. Rhee, "Performance Analysis of IoT-Based Sensor, Big Data Processing, and Machine Learning Model for Real-Time Monitoring System in Automotive Manufacturing," *Sensors,* vol. 18, no. 9, p. 2946, 2018.

[4] J. Bulao, "How Much Data Is Created Every Day in 2021," Tech Jury, 18 May 20021. [Online]. Available: https://techjury.net/blog/how-much-data-is-created-every-day/#gref.

[5] N. Sharma, E. A. Oriaku and N. Oriaku, "Cost and Effects of Data Breaches, Precautions, and Disclosure Laws," *International Journal of Emerging Trends in Social Sciences,* vol. 8, no. 1, pp. 33-41, 2020.

[6] M. T. Tun, D. E. Nyaung and M. P. Phyu, "Performance Evaluation of Intrusion Detection Streaming Transactions Using Apache Kafka and Spark Streaming," *2019 International Conference on Advanced Information Technologies (ICAIT),* pp. 25-30, 2019.

[7] K. Adhinugraha, W. Rahayu, T. Hara and D. Taniar, "On Internet-of-Things (IoT) gateway coverage expansion," *Future Generation Computer Systems,* vol. 107, pp. 578-587, 2020.

[8] H. Insights, "Apache Kafka," [Online]. Available: https://discovery.hgdata.com/product/apache-kafka.

[9] K. M. M. Thein, "Apache Kafka: Next Generation Distributed Messaging System," *International Journal of Scentific Engineering and Technology Research,* vol. 3, no. 47, pp. 9478-9483, 2014.

[10] N Garg, Apache Kafka : set up Apache Kafka clusters and develop custom message producers and consumers using practical, hands-on examples. Birmingham: Packt Publishing, 2013.

[11] H. Wu, Z. Shang and K. Wolter, "TRAK: A Testing Tool for Studying the Reliability of Data Delivery in Apache Kafka," *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW),* pp. 394-397, 2019.

[12] H. Wu, "Research Proposal: Reliability Evaluation of the Apache Kafka Streaming System," *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW),* pp. 112-113, 2019.

[13] J. Kreps, "Project Metamorphosis Month 2: Cost-Effective Apache Kafka for Use Cases Big and Small," Confluent, 17 June 2020. [Online]. Available: https://www.confluent.io/blog/cost-effective-kafka-for-lower-tco/.

[14] N. Jatana, S. Puri, M. Ahuja, I. Kathuri and D. Gosain, "A Survey and Comparison of Relational and Non-Relational Database," *International Journal of Engineering Research & Technology,* vol. 1, no. 6, pp. 1-5, 2012.

[15] D. Damodaran, S. Salim and S. M. Vargese, "Performance Evaluation of MySQL and MongoDB Databases," *International Journal on Cybernetics & Informatics,* vol. 5, no. 2, pp. 387-394, 2016.

[16] A. van der Stock, B. Glas, T. Gigler and N. Smithline, "OWASP Top Ten," The OWASP® Foundation, 1 January 2017. [Online]. Available: https://owasp.org/www-project-top-ten/.

[17] "Hard Drive Data and Stats," Backblaze, 2018. [Online]. Available: https://www.backblaze.com/b2/hard-drive-test-data.html.

[18] "kafkajs - npm," [Online]. Available: https://www.npmjs.com/package/kafkajs.

[19] "dwyl/english-words: A text file containing 479k English words for all your dictionary/word-based projects e.g: auto-completion / autosuggestion," [Online]. Available: https://github.com/dwyl/english-words.

[20] B. H. Chaitra and P. Singh, "Comprehensive Review of Stream Processing Tools," *International Research Journal of Engineering and Technology (IRJET),* vol. 7, no. 5, pp. 3537-3540, 2020.

[21] "Kafka .NET Client," Confluent, 2021. [Online]. Available: https://docs.confluent.io/clients-confluent-kafka-dotnet/current/overview.html.

[22] R. Spurgeon and N. Verbeck, "Best Practices to Secure Your Apache Kafka Deployment," Confluent, 28 May 2020. [Online]. Available: https://www.confluent.io/blog/secure-kafkadeployment-best-practices/.

[23] C.-W. Ten, C.-C. Liu and M. Govindarasu, "Vulnerability Assessment of Cybersecurity for SCADA Systems Using Attack Trees," *2007 IEEE Power Engineering Society General Meeting,* pp. 1-8, 2007.