NoSQL Security Vulnerabilities: A Case Study of Redis

Anne Katherine Fike, Timothy Ross, and Jake Stensrud

***Abstract*—Redis is an in-memory, open-source, NoSQL database that is widely used across the software industry. This paper takes an in-depth look at the Redis database from a cybersecurity perspective, specifically by expounding on CVE records filed against the database. We began by selecting three records that were recent, severe, and complex. Then we began research into the vulnerabilities detailed in the record. In this paper, we report our findings on three CVE vulnerabilities in the Redis Database: CVE-2022-33099, CVE-2022-0543, and CVE-2021-32675. CVE-2022-33099 contains a heap-based buffer overflow vulnerability leading to remote code execution. CVE-2022-0543 features a sandbox escape attack leading to a possible remote code execution as well. Lastly, CVE-2021-32675 details a vulnerability arising from the absence of limitations regarding resource allocation that can lead to a denial-of-service attack. We elaborate on the mechanics behind these vulnerabilities as well as mitigation and prevention strategies. In addition, we cover the related security principles in each CVE’s context as well as other related CVE records pertaining to the three main CVEs of our research scope. For CVE records where it is applicable, we also provide information on the background and history of the CVE record. Overall, our findings concluded that there is a breadth of deficiencies within Redis that expose users to harmful attacks including an absence of checks to ensure Lua scripts are not harmed, a lack of limits imposed on resource allocations, as well as a lack of checks in Redis-CLI for harmful user input. Thus, it is shown that an absence of checks on user input is a common deficiency among Redis systems. Consequently, these absences of checks lead to consequences in confidentiality, integrity, and accessibility of systems.**

1. Introduction

WITH the popularization of NoSQL databases due to its usage by “major internet companies” such as “Google, Amazon, Twitter, and Facebook” because of the limitations of relational databases in solving some modern-day data storage challenges, more companies are adopting this type of database for their data storage systems [1]. However, with this increased popularization and reliance on this technology, this visibility causes more vulnerabilities to be exposed and exploited by interested parties. Thus, it is the goal of our research to examine the common vulnerabilities found in NoSQL databases, highlight how and why they exist, what they expose, and how they can be prevented. With this analysis, we hope to showcase the necessary actions that need to be taken in protecting against these vulnerabilities as we as to improve the knowledge/awareness of the commonality of these vulnerabilities. Our specific scope of research is examining in depth three CVEs relating to NoSQL vulnerabilities found in applications employing Redis databases.

*A. Background on NoSQL Databases*

To begin, we illuminate what NoSQL databases are and how they improved earlier forms of databases. Firstly, the term “NoSQL” was used by Carlo Strozzi in 1998 “to name his lightweight, open-source relational database that did not expose the standard SQL interface” [1]. This name marked the “emergence of a growing number of non-relational, distributed data stores that often did not attempt to provide ACID (Atomicity, Consistency, Isolation, Durability) guarantees” [1]. This type of database was preceded by E.F. Codd’s relational database model in 1970, where “applications [would] search for data by content” in a “table with rows and columns and schema...[that] specific the structure, including the name of relation, name, and type of each column” [1]. However, with NoSQL, you are not required to search for information and organize data by relationships. In addition, NoSQL databases are “highly optimized for retrieve and append operations and often offer little functionality beyond record storage” [1]. Although “full SQL systems” offer increased “run time flexibility” as compared to NoSQL, “this is compensated” in NoSQL “by significant gains in scalability and performance for certain data models” [1]. The categorization of NoSQL databases occurs through identifying “the way [the databases] store data” [1]. The categories of NoSQL databases are “key-value stores, BigTable implementations, document store databases, and graph databases” [1].

For our research, we are concerned with vulnerabilities created utilizing Redis NoSQL databases. For a description of Redis, I reference the Redis company’s self-description of the database displayed on their website: “Redis is an open source, in-memory key-value data structure store, which can be used as a database, cache, or message broker” [2]. To break down what this means, I first define what “open-source” refers to in terms of databases. For an open-source database, this is a database in which “anyone can easily view the source code”; thus, it is “open and free to download” [3]. Because of this availability, “there is a risk of coding malfunction” and, thus, these types of databases “[pose] a significant security risk” [3]. However, this security risk can be mitigated somewhat by the fact that open-source code allows for it to be more visible to developers, which allows for more examination of and fixes for security risks [3]. Following this, “in-memory databases” refers to those databases “that are purpose-built to rely on DRAM for data storage to enable sub-millisecond responses” [2]. This differs from “most NoSQL and SQL databases that store data on disk or SSD/flash memory” [2].  The advantage of this is that for every time a “query” or “update” occurs in a database, only the “main memory” is accessed” which is “much faster than any disk” [2]. For the key-value data structure store, this means that the NoSQL database that utilizes this method of storage is made up of “a collection of key-value pairs” in which a “[k]ey-value can be a string, a number, and an entirely new set of key-value pairs encapsulated in an object” [2]. Thus, overall Redis is made up of many different features, which are necessary to understand in the examination of the vulnerabilities caused by the database.

*B. Current Research on NoSQL Databases and Redis Vulnerabilities*

Research has already been conducted in this field of NoSQL database vulnerabilities, which is pertinent to our current research process in identifying common severe vulnerabilities. To start off, research by Ferrari *et al.* showcased the amount of “misconfigurations [in NoSQL databases] that may expose data to the Internet” through their development of “a tool that automatically scans large IP subnets to detect the exposed services and performs security analyses without storing nor exposing sensitive data” [4]. The scope of their research included investigating the “most popular NoSQL databases” such as MongoDB, Elasticsearch, Redis, and Cassandra” [4]. In total, Ferrari *et al.* “analyzed 67,725,641 IP addresses between October 2019 and March 2020, spread across several CSPs, and found 12,276 misconfigured databases” [4]. The risks identified included “*data leaking*,” where users’ privacy is violated, and “*data tampering of resources stored in the vulnerable databases*,” where web service reputation is at risk [4]. Overall, they were able to identify “742 potentially vulnerable websites linked to misconfigured instances with the write permission enabled to anonymous users” [4].

Our motivation for focusing on Redis versus all other NoSQL databases, like the more popular MongoDB as determined by db-engines.com, is because of current events exposing Redis’s major vulnerabilities [5]. In June of 2018, it was found that out of “72,000 Redis servers available online” at the time, “over 10,000 Redis instances were vulnerable to malicious users because” of their being “left open on the Internet without any authentication system in place” [4, 6]. Of these exposed Redis instances, “over 75% of these servers were featuring an SSH key known to be associated with a malware botnet operation” [6]. This was further investigated by Ferrari *et al.* where they “studied if NoSQL instances were already compromised by the attacks [found in June 2018]...by checking the presence of keywords...related to these attacks in their ’tables’ names,” and “[a]mong the services under analysis, Redis [was] the one that [had] received the highest number of attacks, with a striking 30.3% of compromised instances” [4]. Thus, because of the commonality and severity of this vulnerability found in Redis, we sought to investigate other points of weakness in Redis systems in order to be able to highlight major trends in problems in Redis to be able to in the future strengthen its security and users’ trust in the software.

1. Methods

*A. Scope of Research and Research Question*

For our research project, we plan to investigate vulnerabilities found in Redis databases in order to (1) highlight the origins of the weaknesses found in Redis, (2) to determine whether these vulnerabilities are common or not, and (3) to examine to what extent these vulnerabilities can exact extensive damage to systems. For which specific CVEs we will be covering these are CVE-2021-32626, CVE-2021-36675, and CVE-2021-0543.

*B. Definitions of Terms*

For the specific terminology we will be using in terms of how “vulnerability” is defined, we will be operating under the definition given by the Computer Security Resource Center for “software vulnerability” as “a security flaw, glitch, or weakness found in software code that could be exploited by an attacker (threat source)” [7].

As for what Redis is itself, as defined by the company, it is an “open-source, in-memory data store” that can be used as a “database, cache, streaming engine, and message broker” [8]. Some major features of Redis include “in-memory data structures,” “programmability,” “extensibility,” “persistence,” “clustering,” and “high availability” [8]. For the first feature, namely “in-memory data structures,” Redis “provides a collection of native data types,” such as “strings,” “lists,” “sets,” “hashes,” and much more [8, 9]. For “programmability,” Redis gives users a “programming interface that lets you execute custom scripts on the server itself” [8, 10]. In essence, in Redis this means that you have the “ability to execute arbitrary user-defined logic by the server” [10]. For “extensibility,” a “module API” exists for “building custom extensions to Redis in C, C++, and Rust” [8]. For the term “persistence,” its meaning is that it “keeps the dataset in memory for fast access but can also persist all writes to permanent storage to survive reboots and system failures” [8]. “Clustering” refers to “horizontal scalability with hash-based sharing, scaling to millions of nodes with automatic re-partitioning when growing the cluster” [8]. Lastly, “high availability” means there exists “replication with automatic failover for both standalone and clustered deployments” [8].

For our methods, we utilize analyzing vulnerabilities through CVEs filed against Redis. What CVEs are and how vulnerabilities are categorized and scored requires explanation as we mainly utilize this resource for our investigation. To begin CVE stands for Common Vulnerabilities and Exposures and is a program that is sponsored by the U.S. Department of Homeland Security (DHS) Cybersecurity and Infrastructure Security Agency (CISA) [11]. Essentially the program, as explained on the cve.org website, provides a “list of records containing an identification number, a description, and at least one public reference for publicly known cybersecurity vulnerabilities” [12]. This information from CVE is fed into NVD, or the U.S. National Vulnerability Database, which is “built upon and fully synchronized with the CVE List so that any updates to CVE appear immediately in NVD” [12]. What NVD provides that CVE does not is “enhanced information for each record such as fix information, severity scores, and impact ratings” and “advanced searching features” [12]. NVD is also sponsored by the U.S. Department of Homeland Security (DHS) Cybersecurity and Infrastructure Security Agency (CISA) [12]. Thus, in our investigation, we will be referencing both databases in regard to the vulnerabilities we cover.

*C. Process*

The process we used to answer this question was by first identifying CVEs filed against Redis using websites like cve.mitre.org and stack.watch. The preconceived categories which we used to determine what specific CVEs to focus on, included the CVE’s relative recency, the severity of the CVE, and lastly those CVE’s that we found interesting based on our own studies. Thus, the CVEs that were chosen were CVE-2021-32625, CVE-2021-32675, CVE-2021-0543. Following this, to analyze each vulnerability, we looked at the base score, weakness enumeration (CWE), and vectors of attack. In addition to this, we viewed the specific errors in the code that led to the vulnerability’s existence and or its exploitation. If code could not be viewed, a general description of the error of the code was provided. We also looked at what preventative measures were taken against this vulnerability and how/if developers patched the problem. These data points were generally provided by either the CVE or the NVD program. Following this, we looked for if other vulnerabilities like the current CVE being examined existed in other CVEs to showcase the commonality of the issue within Redis or other contexts. Lastly, we determined if others had explored these particular vulnerabilities in their research. We utilized the dl.acm.org database for relevant scholarly journals and/or conference papers. Thus, with our established method, we hoped to showcase the factors that can lead to severe vulnerabilities in Redis, the commonality of such vulnerabilities in Redis, and how to prevent issues like these from cropping up in the future.

III. Results & Discussion

*A. Memory Corruption Vulnerability in Redis*

In our browsing of the stack.watch website, our group tagged a CVE filed against Redis (CVE-2021-32626) with a high score of 8.8 involving an attack with memory corruption [13]. For explaining what problem occurred in this instance, it was found by Meir Shpilraein that in Redis versions 2.6 or newer, “specially crafted Lua scripts executing in Redis can cause the heap-based Lua stack to be overflowed” because of “incomplete checks for this condition” [1,2]. This issue could lead to “heap corruption and potentially remote code execution” [13]. Overall, this issue was patched in versions 6.2.6, 6.0.16, and 5.0.14 of Redis [14]. In addition, a workaround was found, in case updates could not be made, where an individual can “prevent users from executing Lua scripts” altogether “using ACL to restrict EVAL and EVALSHA commands” [14].

In order to prevent future issues like this from happening again, it is important to understand in-depth how this attack works. In this instance, in order to achieve this, first it must be explained what Lua is, what stack overflow is, and how this can lead to heap corruption and remote code execution. To start, Lua is described as an “embeddable scripting language” that “supports procedural programming, object-oriented programming, functional programming, data-drive programming, and data description” [15]. There are two documented vulnerabilities (CVE-2022-28805 and CVE-2022-333099) regarding a heap-based buffer overflow attack vulnerability exposed by the language itself [16].

For the first CVE filed, it is shown that the vulnerability exists within Lua versions 5.4.0 to 5.4.4 (inclusive) that can lead to buffer overflow [16]. This is achieved through “us[ing] a specially crafted luaK\_exp2anyregup call” which “trigger[s] [a] heap-based buffer overflow” and allows for one to “execute arbitrary code on the target system” [16]. For this vulnerability, there are currently no patches for it and neither is there “any official solution to address this vulnerability” [16]. This is concerning as the base score for this CVE is 9.1, which is labeled as in the critical range [17]. The attack vector of this vulnerability is the network, and this attack affects the confidentiality and the availability of the system. This vulnerability exists due to the issue in the Lua’s code function singlevar in lparser.c from versions 5.4.4 and earlier, where the function “lacks a certain luaK\_exp2anyregup call, leading to a heap-based buffer over-read that might affect a system that compiles untrusted Lua code” [18].

For the second CVE (CVE-2022-33099) filed against Lua in reference to a heap-based buffer overflow attack, this vulnerability “allows a remote attacker to perform a denial of service (DoS) attack” [16]. This is accomplished because of a “boundary error in the luaG\_runerror component” where a “remote attack can send specially crafted data to the application, trigger a heap-based buffer overflow, and perform a denial of service (DoS) attack” [16]. The score for this vulnerability is given a 7.5, which is in the high range [19]. The version of Lua that this attack is available is in versions 5.4.0 to 5.4.4 (inclusive) [16]. The attack vector is the network, and with a denial-of-service attack, the availability of the system is what is affected [19]. There have also been no patches or useful solutions to fix this [16].

For how a heap-based buffer overflow attack works, this is explained in CWE-122---the Common Weakness Enumeration ID for the heap-based buffer overflow weakness type---as an attack “where the buffer that can be overwritten is allocated in the heap portion of memory” [20]. The impacts of such a weakness includes creating denial-of-service-related effects such as crashes, exits, restarts, resource consumption of your CPU, and resource consumption of your memory [20]. This particular impact was shown in CVE-2022-33099). In addition, this weakness can impact not only the availability of the system but also the integrity, confidentiality, and access control of the system with the weakness allowing for the “he “execut[ion] [of] unauthorized code or commands” by “overwrit[ing] function pointers that may be living in memory...[and] pointing it to the attacker’s code” [20]. An attack like this was shown in CVE-2022-28805).

Thus, with this analysis of the vulnerabilities existing within the code for Lua, the vulnerability identified in CVE-2021-32626) can be assumed that it was created unintentionally by developers not creating checks for harmful Lua code. Redis “provides a programming interface” that interacts with Lua scripts by allowing “users [to] upload and execute Lua scripts on the server” [21]. This is achieved through utilizing the “EVAL command” in Redis [21]. An example EVAL command is provided below:

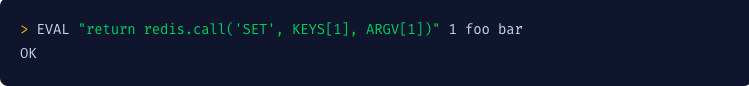


Figure 1: Redis EVAL Command

Source: Adapted from [21]

The user inputs the Lua source code following the “EVAL” keyword. In Lua, the user can call a Redis function, such as calling “SET” to set a key’s value [21]. The “1” represents the number of “key name arguments,” which is represented in Lua with the keyword “KEYS[x]” [21].  Following this parameter are all the arguments necessary for the Lua source code itself. For example, KEYS[1] is getting “foo” and ARGV[1] is getting “bar” [21]. If you do not want to hard code your script each time, you can cache your script by calling “SCRIPT LOAD” followed by your source code in Redis [21]. The next time you want to execute your Lua script, you call “EVALSHA” followed by the “SHA1 digest returned from the server” when you called “SCRIPT LOAD” and the same arguments as you would have provided if you just supplied the source code [21]. This is why one of the methods to mitigate the Lua heap-based buffer overflow vulnerability was to “restrict EVAL and EVALSHA commands” as this is the point of entry users can submit their harmful Lua code [14]. This restriction is achieved through Redis’s “Access Control List,” or ACL. “that allows certain connections to be limited in terms of the commands that can be executed and the keys that can be accessed” [22].

Though the CVE-2021-32626 vulnerability was patched in versions 6.2.6, 6.0.16, and 5.0.14. Redis continues to experience issues with Lua code presenting vulnerabilities for the Redis environment. For example, the vulnerability filed against Redis in CVE-2022-24736 found that “an attack attempting to load a specially crafted Lua script [could] cause NULL pointer dereference which will result with a crash of the redis-server process. This issue at the time “affect[ed] all version of Redis” but it has been fixed in Redis version 7.0.0 [23]. Not only this, but another vulnerability was found (CVE-2022-24735), where “an attacker with access to Redis can inject Lua code that will execute the (potentially higher) privileges of another Redis user” by “exploiting weaknesses in the Lua script execution environment” [23]. Thus, with evidence of these continued vulnerabilities due to Lua, a common vulnerability within Redis exists in the problems that can occur with evaluating problematic Lua scripts. Because of the ability to limit users from calling the command to evaluate Lua scripts, you can easily avoid the issues caused by Lua. However, this issue could be bothersome if you intended to utilize a lot of scripting.

This problem of vulnerabilities created due to Lua scripting is also identified in scholastic research. For example, in Zaki and Indiramma’s research, they seek to “add immense security to a Redis system” [24]. Their reasoning for this was because “Redis does not provide enough security for the data” as “[a]nyone can get the value if the key is known because the data is stored in the form of a key-value pair” [24]. In relation to the vulnerability caused by Lua scripts, they conceded that “[w]hile it would be a very strange use case, the application should not make use of strings obtained from untrusted sources for composing the body of the Lua script” [24]. Thus, though this is not a typical use case, Redis should have in-built security against allowing users to run certain Lua scripts that would create buffer overflow attacks. In addition, there is also research by Sanchez, Bernal, and Parada of CVEs filed against Redis (specifically CVE-2019-10193, CVE-2018-12453, and CVE-2018-12326) of which CVE-2018-12326 involved vulnerability regarding a buffer overflow in “redis-cli of Redis before 4.0.10 and %.x before 5.0 RC3” that “allows an attacker to achieve code execution and escalate to higher privileges via a crafted command line” [25, 26]. It is unclear whether this is a Lua related issue or not; however, it is made clear by [25] that “Redis...is quite vulnerable to NoSQL Injection attacks such as illegal queries” [25]. Thus, through examining outside research, it is clear that vulnerabilities through command line input recently existed and still exist today.

*B. Overview of CVE-2022-0543*

CVE-2022-0543 described a vulnerability in the Redis database system where a flaw in how memory is allocated is known to cause input validation errors [27]. Using the Redis-CLI utility, arbitrary values can be sent that can be used to execute remote code on the server [27]. Thus, attackers need only use a specially made request to carry out this attack. This variation of overflow attack is classified as a sandbox-escape attack [27]. It should be noted that sandbox-escape attacks are quite prevalent in NoSQL database systems [27]. This vulnerability was reported by Reginaldo Silva, a Redis contributor, in January of 2022 [27]. By this point, the vulnerability was already being abused by the Muhstik Malware which used it primarily to grow their botnet [27, 30]. This vulnerability shares many similarities with the Log4J vulnerability, so much of the software that was vulnerable to Log4J was also targeted using this vulnerability [27]. Shortly after its discovery, the US Cybersecurity and Infrastructure Agency (CISA) released an estimation that upwards of 2000 Linux servers were affected, although they admitted that potentially up to 30000 could have already been infected [27]. The confidentiality, Integrity, and availability of infected systems are completely compromised. Due to the danger level of this vulnerability, the threat rating the vulnerability was given was 10, the highest possible CVSS score. Since the discovery of CVE-2022-0543, updates have been made available to Redis that help mitigate the risk of it being abused, however there are still many systems that have not been updated and are still at risk [27].

A sandbox, in the context of the sandbox escape attack, is the combination of an environment that is used to contain a computer process, and that environment’s host [28]. Therefore, a sandbox escape attack is where a user or some process can break free from their sandbox and access any outside resource without being constrained by the rules of the sandbox. When exiting the sandbox, the user or process obtains the privileges of the host. Some common examples of sandboxes are virtual machines, docker containers, and mobile apps, to name a few [28]. There are many techniques that can be employed to make malicious use of this attack, one such being the commanding and controlling of a botnet [28,29]. This begins by executing remote code that when ran contacts websites or IP’s that inject the attacker’s malware into the sandbox [28].

Botnets are a type of malware that is comprised of many infected computers that act collectively as a network and can be controlled remotely. [29]. The number of Botnet based DDoS attacks has increased harshly in the last decade, which is due to the number of insecure devices and software also harshly increasing [30]. Botnets are also the most prevalent source of DDoS attacks on the Internet, even though not all botnets are used for DDoS attacks [30]. The infamous Mirai botnet’s source code was also released to the public. This has since spurred many imitators who have enacted large-scale DDoS attacks against major companies like GitHub and Amazon’s AWS [30]. It is already somewhat easy to get infected by a botnet, as usually, it is a result of clicking on a bad link or opening a suspicious email attachment. In the case of the Redis vulnerability, it is all but guaranteed that an attacker can get their malware onto the victim's system. It is as simple as performing the sandbox escape, then remotely running code that fetches the malware and installs it into the Redis database. Therefore, vulnerable Redis systems are a prime target for attackers who wish to grow their botnet. There are three main means of defending a system from bots or at the very least, means of detecting a bot infection: Signature-based, Anomaly-based, and Specification-based detection [37]. Signature-based detection compares network traffic with rules given by a separate database to detect bots. Anomaly-based detection takes a more machine-learning approach. It analyses normal behavior on the system and creates a profile to compare network traffic to detect bots [37]. Lastly, Specification-based detection searches through internet traffic and compares it to set rules that the user defines [37]. The Washington State Office of the Attorney General has launched “The national cybersecurity education and awareness campaign.” This campaign has released a set of rules that advise the public on ways to stay safe from botnet malware. Their advice is to keep software current, enable automatic updates, protect all internet-connected devices (not just computers), scan all external media, delete any online communications that look suspicious, get savvy about Wi-Fi hotspots, and be cautious about scareware [29]. The last entry of that list refers to the technique that hackers use to get information by falsely informing a user that their system is infected and that they must buy the attacker's software to help remove it. The first rule of that list applies to the Redis database, as updates have been released that help to mitigate the threat of the CVE-2022-0543 vulnerability [27].

One of the most well-known abusers of the CVE-2022-0543 vulnerability is the Muhstik Malware Group. The Muhstik group is thought to originate from China and is responsible for creating one of the most prolific botnets in recent years [31]. Reports of Muhstik malware have been made for around the last 5 years, and a newer iteration has been expanding the botnet through compromised Redis systems [31]. Infected Redis systems will have Muhstik’s botnet binaries run remotely. Once infected, the bot will connect to an internet relay chat server to receive its function [31]. Its function will typically involve downloading files, brute forcing SSH credentials to allow for the running of shell commands and conducting its attacks [31]. The Muhstik Botnet monetizes its exploits by either implementing ransomware, installing cryptocurrency coin miners, or DDOS attacks. In all cases, the botnet also self-propagates [31].

C. Overview of CVE-2021-32675

CVE-2021-32675 is a Denial of Service attack that exploits the Redis Standard Protocol, or RESP. It was patched in Redis versions 6.2.6, 6.0.16 and 5.0.14, and has a CVE severity rating of 7.5 [32]. It can be performed by a malicious user without any authentication or privileges as long as they can reach the Redis database server. It takes advantage of the fact that RESP does not limit the size of requests in any way, allowing an attacker to create a crafted request that when unpacked will consume much if not all the memory allocated for the database instance. It was eventually fixed by limiting the size of requests that can be sent while unauthenticated. This section will examine the specifics that caused the vulnerability, the fixes and mitigation for the vulnerability, the specifics of the patch for the vulnerability, the security principles violated that caused this vulnerability, and other possible fixes.

In order to understand the mechanics behind CVE-2021-32675, a brief summary of the RESP protocol is necessary. RESP is a request-response type protocol that utilizes TCP [33]. It can send simple strings, errors, integers, arrays, and bulk strings, indicated by various special characters at the start of the message. An array is also known as a multi-bulk. In order to send arrays and bulk strings, you must declare the size of them before they are read. For example, for the bulk string “hello world”, you would have:

"$11\r\nhello world\r\n"

$11 indicates that the length of the string is 11 characters and is followed by the string. Return-newlines are used to delimit the protocol in general. Similarly, an array is indicated via a ‘\*’ character, followed by the length, then the delimiter, then the content. CVE-2021-32675 takes advantage of this client-side declaration of lengths to consume substantial amounts of memory. Specifically, in vulnerable version of the software multi-bulks could contain 220 512 megabyte bulks, resulting in a maximum total size of 512 terabytes. This does not necessarily mean that Redis will load 512 terabytes into memory, but it does provide a good understanding of possible scale of these commands: massive. However, the size of these bulks and multibulks was not the only issue that contributed to the full vulnerability. The authentication method for the database caused problems as well.

Redis authentication works by sending an “Auth” command over RESP with a username (before Redis 6.0.0) or username and password (after Redis 6.0.0) [34]. This information is sent unencrypted and may be sent as many times as one pleases without negative repercussions, like being denylisted. More importantly, like most Redis commands this command utilizes bulks and multibulks, without any special mechanisms limiting the size. This means you can launch a denial-of-service attack on Redis by repeatedly sending a large authentication multibulk. Furthermore, you can easily transform this attack from a DOS to a DDOS attack by having multiple unauthenticated clients sending bad Auth commands at once.

It is also worth bringing up that since the authentication method is unencrypted by default, it is possible to steal user credentials if you are capable of eavesdropping on the network [24]. Once many users have had their credentials intercepted, you could conduct a DOS or DDOS attack without even utilizing the vulnerability of the authentication procedure. Simply authenticate the database and begin sending large bulks and multibulks in commands as a valid user. The significance of this vector of attack will become more apparent later when the currently existing solutions are evaluated..

Current suggested fixes for this vulnerability are updating Redis to a fixed version, utilizing integrated support for TLS, or preventing unauthorized connections to Redis via an alternate security mechanism such as a firewall [35]. Using an external security mechanism such as a firewall is a practical solution but does not address the underlying issue, the vulnerability. It only moves the application into a space where it is hopefully isolated against any malicious actors, shifting the burden of security. Integrated support for TLS allows clients to authenticate via a cryptographic exchange utilizing certificates. This switches the authentication method used by Redis to a far more secure alternative. The fixed version of the software addresses part of the vulnerability but does not address the full vulnerability.

In the fixed version, Redis’ solution to the issue was a simple one. Redis now limits the amount of data an Auth command can send from 220 512 megabyte bulks to 10 16-kilobyte bulks [36]. This prevents an unauthenticated user from attacking the system or at least makes an attack far more resource intensive. However, this fix does not prevent an authenticated user from sending substantial amounts of data that impact performance, which means an attacker or malicious user could use authenticated accounts to compromise the availability of Redis. It is difficult to make a database system where an authenticated user cannot use a large number of resources without compromising necessary system features. Redis likely weighed their options and left this vector of attack, seeing it as an acceptable risk. This does not change the fact that authenticated users can still consume exceptionally large amounts of data within Redis, which is described as part of the vulnerability. It is not unreasonable to suspect an attacker could gain access to user credentials given how insecure the authentication method is. It is also not unreasonable to suspect that if an attacker did gain access to user credentials, there would be security issues that would trivialize this vulnerability.

It is possible to fix the vulnerability by examining which principles of security were violated in the first place. The issue behind this vulnerability can be thought of as a violation of the principle of least privilege. When authenticating or using any other feature, users should only be given access to the resources they require and are permitted to use. When Redis allows a user to send hundreds of terabytes for a simple authentication, they are allowing the user to consume far more resources than they would ever need for that operation. Permitting an authenticated user to send many large requests over multiple connections can also impact performance and is allotting them more resources than should be allowed. It follows that the solution to the vulnerability would contain some way to manage the number of resources unauthenticated and authenticated users are utilizing.

For authenticated users, this is simple in theory. The database should specify the number of resources a user can utilize at peak usage to prevent authenticated users from impacting the software’s availability. A priority by which users split available resources should also exist, such that users with higher privilege can supersede the resource needs of users with lower privilege, and users with equal privilege share resources. Resource usage should be logged so that it can be monitored, and users who attempt to use more resources than the policy permits should have their attempt recorded and their attempt failed. Furthermore, the size of certain commands like authentication should be restricted to more reasonable sizes. Notably, this restriction is already present to a small extent in the patch for the vulnerability.

For unauthenticated users, it is much more difficult to restrict resource usage without locking out other users. Since we cannot distinguish between users, we have no way of knowing who is a malicious attacker and who is a regular user. The best recourse would seem to be a combined solution of minimizing the number of resources unauthenticated users can access and implementing security measures outside of the database system, such as the suggested TLS implementation and firewall. Unauthenticated users should be allocated a small slice of the system resources to work with, such that they will not interfere with the requests of authenticated users. Another larger slice of resources should be allocated for authentication alone such that users can always authenticate, and authentication should be relatively small in its resource usage.

While these ideas sound simple in theory, it is likely they would be difficult to implement in practice. Care would have to be taken to introduce proper policy enforcement and select proper values for resource allocation. Given the possible complexity of implementation, great care would also be needed to not introduce more vulnerabilities into the system. However, if implemented properly, these fixes would mitigate or patch CVE-2021-32675 in its entirety.

IV. Conclusion

Due to the popularization of NoSQL databases, there has been more interest by cyber-criminals to find and exploit vulnerabilities in them. Redis has an exceptional number of vulnerabilities, some of the most notable are documented by the reports CVE-2022-33099, CVE-2022-0543, and CVE-2021-32675. These describe three recent, complex, and severe vulnerabilities to the Redis database. We have gone in-depth into these reports to bring to light the severity of these vulnerabilities, but also preventative measures against them. CVE-2021-32675 is related to a vulnerability that can cause memory corruption that can allow Lua scripts to overflow the heap and allow remote code execution. This vulnerability was patched, but a workaround was found.  After the evaluation of similar attacks, it was inferred that this vulnerability could be rendered useless if the database could properly evaluate the Lua scripts it was given so that it could identify problematic code. CVE-2022-0543 describes a vulnerability that is caused by input validation errors. This allowed attackers to execute a sandbox escape to gain root-level access. The vulnerability was most commonly targeted by botnet-related DDoS attacks. Dissimilarly to the last CVE report, an update was released that targeted this vulnerability and as of the writing of this report, no meaningful workarounds have been found. CVE-2021-32675 describes a vulnerability that enables a DoS attack to be enacted that exploits the Redis Standard Protocol.  An attacker using this vulnerability could use all the memory allocated for a server by entering a massive input to RESP, which would crash the system. Patches to this vulnerability have since been issued. All the issues we have uncovered lead to the conclusion that NoSQL databases, notably but not exclusively Redis, can be debilitated by their vulnerabilities. Memory security and input validation will continue to be the main target for attackers to try and exploit due to the access that can give. This should send the message that the developers of not only NoSQL databases, but any software, should thoroughly ensure that oversights such as those in the reports described cannot occur in their software.

For future research purposes, it would be beneficial to examine these issues we have discovered in Redis and investigate if other forms of NoSQL database software experience similar issues. In this way, it could be determined if these vulnerabilities are a common trend among NoSQL or just among Redis. The same research method we employed in our research could be utilized in this future research of other NoSQL databases. The databases that would we be interested in further investigating would be MongoDB, Cassandra, and ElasticSearch as these types of NoSQL databases are relatively popular.

In addition to this, comparing what common vulnerabilities are found in among document databases, key-value stores, column-oriented databases, and graph databases would be beneficial in understanding if storage type determines the severity and commonality of particular vulnerabilities. In analyzing NoSQL databases through this scope, developers can become more informed at the particular risks they are taking on in utilizing one type of NoSQL database management system over another.

Lastly, to answer the question as to whether to vulnerabilities we discovered in Redis are also seen in SQL databases, further research would need to be conducted. This would also be useful in determining if not only Redis, but also SQL databases frequently are exposed to the same attacks. It would also be useful to compare what common countermeasures are taken in NoSQL vulnerabilities versus SQL vulnerabilities to see if the two database management types have any underlying qualities that makes one software more apt to deal with particular security threats than the other.

Thus, with all of these proposed research ideas, if these were to be carried out, information about the security vulnerabilities of not just Redis, but also NoSQL databases in general as well as SQL databases can be better understood. Even more, with this analysis readily available, developers of Redis and other NoSQL databases as well as customers who are deciding with database to utilize in their projects will be well-equipped with a breadth of analysis on the different types of vulnerabilities found in these databases in addition to the severity and commonality of the issues. This can also help prevent and mitigate the cost of damages due to lack of preparation and ignorance to specific weaknesses.

References

[1] K. L. Berg, T. Seymour, and R. Goel, “History of Databases,” *International Journal of Management & Information Systems (IJMIS)*, vol. 17, no. 1, pp. 29–36, Dec. 2012.

[2] “What Is NoSQL,” *Redis*, 03-Oct-2022. [Online]. Available: https://redis.com/nosql/what-is-nosql/#:~:text=What%20is%20Redis%20NoSQL%3F,It's%20a%20NoSQL%20database. [Accessed: 08-Dec-2022].

[3] “Difference between Open Source Database and Commercial Database,” *GeeksforGeeks*, 06-Jun-2022. [Online]. Available: https://www.geeksforgeeks.org/difference-between-open-source-database-and-commercial-database/. [Accessed: 08-Dec-2022].

[4] D. Ferrari, M. Carminati, M. Polino, and S. Zanero, “NoSQL Breakdown: A Large-scale Analysis of Misconfigured NoSQL Services,” *Annual Computer Security Applications Conference*, pp. 567–581, Dec. 2020.

[5] “Engines Ranking,” *DB-Engines*. [Online]. Available: https://db-engines.com/en/ranking. [Accessed: 08-Dec-2022].

[6] C. Cimpanu, “Around 75% of Open Redis Servers Are Infected With Malware,” *BleepingComputer*, 01-Jun-2018. [Online]. Available: https://www.bleepingcomputer.com/news/security/around-75-percent-of-open-redis-servers-are-infected-with-malware/. [Accessed: 08-Dec-2022].

[7] “Software Vulnerability - Glossary | CSRC,” *NIST Computer Security Resource Center*. [Online]. Available: https://csrc.nist.gov/glossary/term/software\_vulnerability#:~:text=Definition(s)%3A,an%20attacker%20(threat%20source). [Accessed: 08-Dec-2022].

[8] *Redis*. [Online]. Available: https://redis.io/. [Accessed: 08-Dec-2022].

[9] “Redis data types,” *Redis*. [Online]. Available: https://redis.io/docs/data-types/. [Accessed: 08-Dec-2022].

[10] “Redis programmability,” *Redis*. [Online]. Available: https://redis.io/docs/manual/programmability/. [Accessed: 08-Dec-2022].

[11] “Home | CVE,” *CVE*. [Online]. Available: https://www.cve.org/. [Accessed: 08-Dec-2022].

[12] “Qs | CVE,” *CVE*. [Online]. Available: https://www.cve.org/ResourcesSupport/FAQs#pc\_introcve\_nvd\_relationship. [Accessed: 08-Dec-2022].

[13] “CVE-2021-32626,” *NVD*, 04-Oct-2021. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2021-32627. [Accessed: 08-Dec-2022].

[14] Y. Gottlieb, “Lua scripts can overflow the heap-based Lua Stack,” *GitHub*, 2021. [Online]. Available: https://github.com/redis/redis/security/advisories/GHSA-p486-xggp-782c. [Accessed: 08-Dec-2022].

[15] “Lua: About,” *Lua*. [Online]. Available: https://www.lua.org/about.html. [Accessed: 08-Dec-2022].

[16] “Heap-based buffer overflow in Lua,” *Cybersecurity Help*, 11-Apr-2022. [Online]. Available: https://www.cybersecurity-help.cz/vdb/SB2022041112. [Accessed: 08-Dec-2022].

[17] “CVE-2022-28805,” *NVD*, 08-Apr-2022. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2022-28805. [Accessed: 08-Dec-2022].

[18] “singlevar in lparser.c in Lua through 5.4.4 lacks a...,” *GitHub*, 08-Apr-2022. [Online]. Available: https://github.com/advisories/GHSA-pxhp-rhgc-5jx8. [Accessed: 08-Dec-2022].

[19] “CVE-2022-33099,” *NVD*, 2022. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2022-33099. [Accessed: 08-Dec-2022].

[20] “CWE-122: Heap-based Buffer Overflow,” *CWE*. [Online]. Available: https://cwe.mitre.org/data/definitions/122.html. [Accessed: 08-Dec-2022].

[21] “Scripting with Lua,” *Redis*. [Online]. Available: https://redis.io/docs/manual/programmability/eval-intro/#:~:text=Redis%20lets%20users%20upload%20and,from%20scripts%20is%20very%20efficient. [Accessed: 08-Dec-2022].

[22] “ACL,” *Redis*. [Online]. Available: https://redis.io/docs/management/security/acl/#:~:text=The%20Redis%20ACL%2C%20short%20for,a%20valid%20password%20to%20authenticate. [Accessed: 08-Dec-2022].

[23] “Redis 7.0 release notes,” *GitHub*, 2022. [Online]. Available: https://raw.githubusercontent.com/redis/redis/7.0/00-RELEASENOTES. [Accessed: 08-Dec-2022].

[24] A. K. Zaki and M. Indiramma, “A novel redis security extension for NoSQL database using authentication and encryption,” *2015 IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT)*, Mar. 2015.

[25] R. A. G. Sánchez, D. J. M. Bernal and H. D. J. Parada, “Security Assessment of NoSQL, MongoDB, Redis, and Cassandra Database Managers,” *2021 Congreso Internacional de Onnovación y Tendencias en Ingeniería (CONIITI)*, 2021, pp. 1-7, doi: 10.1109/CONTIITI53815.2021.9619597.

[26] “CVE-2018-12326,” *NVD*, 17-Jun-2018. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2018-12327. [Accessed: 08-Dec-2022].

[27] A. Pantelli, “Redis Vulnerability CVE-2022-0543,” *Packt*, 10-Jun-2022. [Online]. Available: https://security.packt.com/redis-vulnerability-cve-2022-0543/. [Accessed: 08-Dec-2022].

[28] A. Herd, “Sandbox escapes,” *Pentesting KB 4 Techno Herder*, 10-Jun-2021. [Online]. Available: https://hack.technoherder.com/sandbox-escapes/#:~:text=A%20sandbox%20escape%20is%20any%20type%20of%20exploit,interface%20%28shell%29%20for%20the%20guest%20to%20operate%20in. [Accessed: 08-Dec-2022].

[29] “Botnet Facts,” *Washington State Office of the Attorney General*. [Online]. Available: https://www.atg.wa.gov/botnet-facts. [Accessed: 08-Dec-2022].

[30] F. Hussain *et al*., "A Two-Fold Machine Learning Approach to Prevent and Detect IoT Botnet Attacks," in *IEEE Access*, vol. 9, pp. 163412-163430, 2021, doi: 10.1109/ACCESS.2021.3131014.

[31] L. Howler, “Muhstik botnet targeting Redis servers using a recently revealed vulnerability in the database system,” *HowToRemove.Guide*, 28-Mar-2022. [Online]. Available: https://howtoremove.guide/muhstik-botnet-targeting-redis-servers/. [Accessed: 08-Dec-2022].

[32] “CVE-2021-32675,” *NVD*, 04-Oct-2021. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2021-32675. [Accessed: 08-Dec-2022].

[33] “RESP protocol spec,” *Redis*. [Online]. Available: https://redis.io/docs/reference/protocol-spec/. [Accessed: 08-Dec-2022].

[34] “Auth,” *Redis*. [Online]. Available: https://redis.io/commands/auth/. [Accessed: 08-Dec-2022].

[35] Y. Gottlieb, “DOS vulnerability in Redis,” *GitHub*, 04-Oct-2021. [Online]. Available: https://github.com/redis/redis/security/advisories/GHSA-f6pw-v9gw-v64p. [Accessed: 08-Dec-2022].

[36] “Prevent unauthenticated client from easily consuming lots of memory (...,” *GitHub*, 04-Oct-2021. [Online]. Available: https://github.com/redis/redis/commit/5674b0057ff2903d43eaff802017eddf37c360f8. [Accessed: 08-Dec-2022].