RESEARCH BLOG

Research by:

Daniel Corak, Julian-Ferdinand Vögele and Vasileios Mitrousis (External)

Chaining Three Zero-Day Exploits in ITSM Software ServiceTonic for Remote Code Execution

November 2, 2021

In a recent Red Team exercise at a market-leading multinational, we **chained three zero-days** found in our client's IT service management (ITSM) software (ServiceTonic) to **get unauthenticated remote code execution**. First, we exploited an **HQL injection vulnerability to retrieve relevant data** from the software's database (Step 1). We then utilized a **vulnerability in the SSO (single-sign-on) implementation** (Step 2) and **combined it with the retrieved data** (Step 3) to log in as an admin. Finally, we found **a path traversal vulnerability**, used it to **upload a web shell** (Step 4), and got **remote code execution** (Step 5).

What makes this attack so interesting is that it illustrates the risk companies face by not enforcing rigid security standards against 3rd party software products before deploying them in their networks. Less known but widely used software products tend to be particularly susceptible as they generally undergo less scrutiny by the security community. This blogpost explains the vulnerabilities found and discusses how they were exploited.

Hacking target: Organizations using ServiceTonic as service management solution

ServiceTonic is a multi-purpose service management software for companies of all sizes with four main solutions: asset management, customer service, IT service management (ITSM), and enterprise service management (ESM). The vulnerabilities were found in ServiceTonic's ITSM solution (up to version 9). According to the vendor website, ServiceTonic products are used by Fortune 500 companies.

Decompiling ServiceTonic shows initial flaws and suggests deeper analysis

Our initial probing of the software's Internet-accessible login window with basic attacks, such as default/common credentials or basic SQL injections, were unsuccessful. Undeterred, we downloaded ServiceTonic's on-premise version, installed it locally, and decompiled the Java application.

What is an HQL injection?

The logs of the locally deployed instance revealed that ServiceTonic makes use of Hibernate ORM (or simply Hibernate), an object–relational mapping tool for Java. In object-oriented systems, there is a discrepancy between the object model (which stores data as graphs) and relational databases (which store data in tabular format). Hibernate facilitates the mapping of those Java objects to database tables.

Hibernate makes use of a query language called HQL that is similar in appearance to SQL. Compared with SQL, however, HQL is fully object-oriented (i.e., it understands notions like inheritance, polymorphism, and association) and operates on objects instead of tables and columns. HQL queries are translated into database-specific queries before passing them to the underlying relational database, making them independent of the database syntax. Given HQL's own syntax and its tendency to be more restrictive (e.g., there exists no way to query unmapped tables), regular SQL injections do not work. However, as shown by other researchers, it is possible to escape the HQL context and enter the SQL context to communicate directly with the database by injecting specially crafted HQL queries (Figure 1). This is referred to as an HQL injection attack.

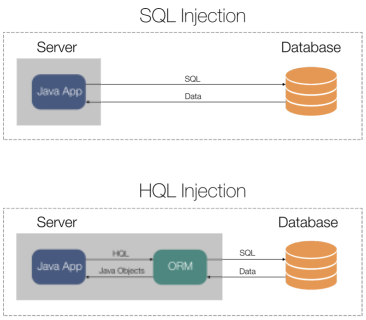

Figure 1: SQL injection vs. HQL injection

Step 1: Locate HQL injection to retrieve data from database (CVE-2021-28022)

Common escape characters and injection strings in the login window did not allow for an SQL injection. The frontend simply returned "Please review your user and password and try again" for all attempts. After deploying ServiceTonic locally, we looked for hints in log files and found that a single quote character triggered following error message:

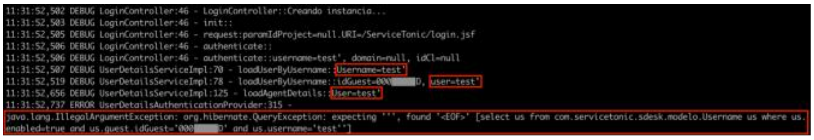

Figure 2: Log with username input and corresponding error message

This error message did not only reveal that an injection was possible, but that it required an HQL injection instead. Looking at the corresponding decompiled code confirmed the injection and trivial logical injections provided a practical proof. The HQL query that caused the error is shown in Figure 3.

```
SELECT us
FROM   Username us
WHERE  us.enabled = true AND us.guest.idGuest = "'" + idGuest + "'" AND us.username = "'" + username + "'"
```

Figure 3: Vulnerable HQL query used for checking the username

The injection occurs due to missing sanitization of the username, which is controlled by the attacker through the login window. Generally, for injections to work, one must be able to distinguish between true and false responses (e.g., does a password start with a specific letter or not?). There are two main approaches: through **response content** (e.g., error messages in the frontend, HTTP response codes and/or lengths) or a side channel (e.g., response time). In our case, only the time-based approach was possible. Such time-based injections are used in scenarios where there are no differences in the response content. The attacker needs to send queries to the database that force an execution delay in the database engine and thus response time.

Figure 4 shows an example of how a time-based HQL injection allows someone to retrieve a username by escaping the HQL context through a **magic function**. These magic functions vary depending on the database (see Figure 5). They work because HQL accepts functions that are not part of the HQL specification and assumes that they are executable by the underlying database.

available, heavy queries can be used. These take noticeable time to get executed by the database engine (e.g., heavy cross-join).

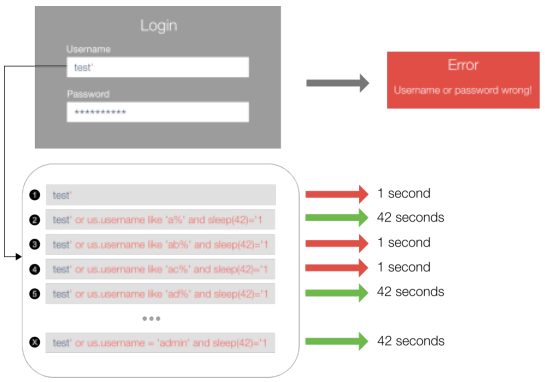

Figure 4: Simple time-based HQL injection example

DBMS	Magic function	Notes
MySQL	sleep(42)	/
Oracle DB	NVL(TO_CHAR(DBMS_XMLGEN.getxml('select 42'))	/
MSSQL	LEN([U+00A0](select[U+00A0]42))	MSSQL allows Unicode delimiters, such as U+00A0 between SQL tokens. On the other hand, HQL allows Unicode characters in identifiers of functions and parameters. When using e.g. U+00A0 as delimiters for a query, HQL will interpret it as function, but in MSSQL it will be interpreted as a valid query

Figure 5: DBMS-specific magic functions

Note that a login-bypass was not possible as only the username field on the login page is injectable and the password is checked separately.

Step 2: Locate SSO functionality (CVE-2021-28024)

While the HQL injection vulnerability (CVE-2021-28022) helped us to retrieve all kinds of data from the database, ServiceTonic stores passwords as hashes (i.e., not as cleartext). For strong passwords, cracking these hashes is computationally infeasible. Therefore, we dug deeper into the login-related code and stumbled across an SSO (single sign-on) functionality. We found that for SSO a separate password is created by ServiceTonic, which always follows the same pattern:

```
base64(sha1(username + current date (yyyyMMddHH) + download code)) + "_SSOA"
```

With this in mind, we concluded that we **do not need to know or crack an existing password,** but we can simply **generate** a new SSO password for a specific user by:

1. Concatenating their username, the current date, and a random, but low entropy download code
2. Applying the SHA1 algorithm to it
3. Converting it into base64
4. Appending the string _SSOA to the result

The vulnerability (CVE-2021-28024) thus lies in the **limited randomness** when creating SSO passwords (i.e., username is often not hard to figure out, current date can be generated, and download code has a low complexity, is the same for all users and follows a simple pattern).

Step 3: Combine CVE-2021-28022 and CVE-2021-28024 to log in as admin

Combining the HQL injection vulnerability (CVE-2021-28022) and the flawed implementation of the SSO functionality (CVE-2021-28024) made it straightforward to determine the SSO password for an admin user:

1. We exploited CVE-2021-28022 to retrieve a cleartext admin username by querying the user table with IS_ADMINISTRATOR set to true (i.e., IS_ADMINISTRATOR=1). Assuming the username has a maximum length of 10 upper- and lower-case characters (10*52), without further optimization, the worst-case for determining it is **520 requests**. Using divide-and-conquer, this number can be reduced to **~60 requests**.
2. We exploited CVE-2021-28022 again to retrieve the cleartext download code by querying the respective table. Downloading the ServiceTonic software multiple times had shown that the download code always consists of 8 numbers and 1 uppercase (e.g., 00012911D) with the number increasing by 1 per download (i.e., 00012911D, 00012912D, ...). Using this information and without further optimization, the worst-case for determining the download code is **68 requests** (0 (position 1-3 are 0) + 2 (position 4) + 10 * 4 (position 5-8) + 26 (position 9)). Using divide-and-conquer, this number can be reduced to **~20 requests**.

Based on the assumptions about the username, the worst-case for determining the SSO is without optimization is **582 requests** (520 + 62) (and **only ~80 requests using divide-and-conquer**). If you are generally interested in exploit optimization, check out one of our previous blogpost.

Step 4: Upload a web shell and ensure it can be access through a website using a path traversal vulnerability (CVE-2021-28023)

Having admin access to the service management software of a company is a great first step from an attacker's perspective. However, to get more than just sensitive information (e.g., social security numbers) and to compromise the whole network, remote code execution is needed.

Through an internal review of the portal, a ServiceTonic feature was found that allows admins to install services by uploading zip files. The extraction mechanism for such zip files turned out to be vulnerable to a path traversal attack by specifying relative path names (e.g., ../../public/webshell.jsp). By uploading a specially crafted zip file containing our web shell, the web shell could be placed in a publicly reachable folder.

Step 5: Remote code execution

Through the web shell, arbitrary commands can be executed with system privileges on the server. Note that no privilege escalation was required since the ServiceTonic software was executed with system privileges. This allowed us to move laterally and fully compromise the network.

Conclusion: Trust is good, testing is better

In this blogpost we demonstrated how three zero-days in one software product were chained together to access a company network through an Internet-accessible web application and then fully compromise it. While detecting the zero-days and chaining them together required perseverance, reverse engineering skills, and creativity, the vulnerabilities themselves are rather basic in their nature.

The attack clearly shows the potentially devastating effects of the lack of basic security enforcement when acquiring third-party products. Less known, but often widely used software products tend to be particularly susceptible as they generally undergo less public scrutiny, which has also been shown by recent events (e.g., Kaseya, Solarwinds).

To reduce the risk, companies should understand how 3rd party software solutions could be leveraged as an attack vector and should try to find security solutions that acknowledge this as a risk and try to mitigate it. Penetration tests are not not enough if their tests are not expanded according to the changing technology landscape. For example, while HQL has existed for many years, testing for HQL injections it is still not common practice.

RECOMMENDED

Your Blockchain is only as secure as the application on top of it

Applications interacting with blockchain networks can be an attack surface to malicious actors and therefore need to be reviewed thoroughly.

RECOMMENDED

Achieving Telerik Remote Code Execution 100 Times Faster

A cryptographic vulnerability in the development software Telerik UI from 2017 turned out to be impractical to exploit until now. This blogpost details the optimization techniques deployed, which can be applied to similar issues in other software.

RECOMMENDED

The physical access control market is ripe for an upgrade to modern technology

Physical access control systems today predominantly use access badges with weak cryptography or no cryptography at all despite better building blocks being available.

Imprint

Security Research Labs GmbH
Brunnenstrasse 181
10119 Berlin

Registration. HRB 128449
District court. Berlin-Charlottenburg
EU-VAT. DE 815 218 931
Managing director: Karsten Nohl