WHITE PAPER

Breaking IBM WebSphere Authentication by Abusing Crypto Flaws in LTPA Tokens

(CVE-2022-22475 & CVE-2022-22476)

Tom Tervoort

Secura
A BUREAU VERITAS COMPANY

Secura
A BUREAU VERITAS COMPANY

Breaking IBM WebSphere Authentication by Abusing Crypto Flaws in LTPA Tokens

(CVE-2022-22475 & CVE-2022-22476)

Nowadays, web applications often rely on cryptographically protected tokens to facilitate single sign-on or maintain sessions across distributed servers. Such tokens contain expiration dates and the identity of the current user, and are stored in the user's browser. It is essential that these users are not able to change the contents of these tokens, as that could allow, for instance, impersonation of other user, elevation of privileges or authentication bypasses. A crypto bug in a token implementation can lead to multiple forms of authentication vulnerabilities.

During my analysis of several token implementations, I found that the way that the application server IBM WebSphere Liberty had implementation flaws in its implementation of the Lightweight Third Party Authentication (LTPA) protocol, a cryptographic token scheme used by multiple IBM products. By combining this implementation bug with cryptographic weaknesses in the protocol itself, an attacker could to change their token into one belonging to any other user. While this attack involves a tricky adaptive chosen-ciphertext attack, it can be easily automated with a script that usually only takes a few seconds to execute. I also found a second (less severe) impersonation attack involving the injection of a delimiter character.

Like Zerologon and many SAML vulnerabilities, this is another example of an exploitable authentication vulnerability caused by flaws in (in implementation of) a somewhat obscure but widely used cryptographic protocol from the early 2000's. This is probably also not the last bit of obscure legacy cryptography relied upon for critical functionality, highlighting the need for more research in this area.

The LTPA Protocol

Lightweight Third Party Authentication (LTPA) is a mechanism used by (ex-)IBM products such as WebSphere and Lotus Domino to facilitate single sign-on and stateless session management. There are two versions of this protocol. My research focuses on the "**more secure**" version 2 and its use in IBM WebSphere Liberty, a modern application server for Java web apps.

LTPA version 2 basically just offers an encrypted cookie, called **LtpaToken2**, which embeds a user identifier and an expiration date. The key material used to both create and validate these cookies is stored in a file named **ltpa.keys.**

This file can be shared with different web servers, allowing each of them to either issue or validate these cookies (and therefore the client's identity) without having to do a lookup to some centralized database to check a session identifier. Keeping these keys secret is critical to being able to discern the authenticity of the tokens.

The ltpa.keys file contains two types of keys: an AES key for encryption and an RSA key pair for signing. Together these are used to compose a signed and encrypted token, in the manner shown by Figure 1.

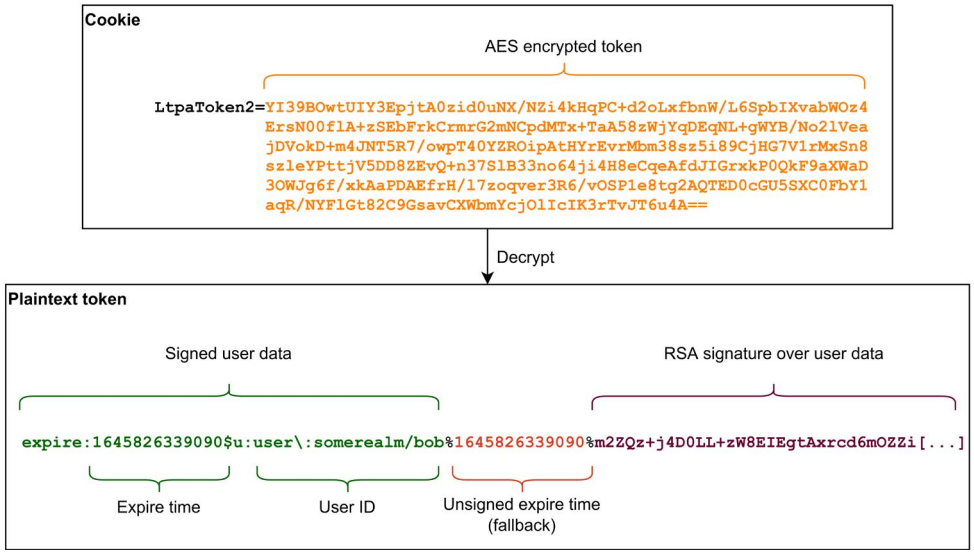


Figure 1: Structure of a typical LTPA token.

Validation of such a token consists of the following steps:

1. Use the AES key from the ltpa.keys file to decrypt the cookie (see the next section for details). The token is invalid if decryption fails.
2. Split the result on % symbols in three parts: **user data**, an **unsigned expiration time** and a **signature**. The token is invalid if there are not exactly three percent signs in it.
3. Parse the user data string into a key-value mapping.
4. Look up the expiration time (as a POSIX timestamp) from the user data using the "expire" key. If not present, use the unsigned expiration time instead. The token is invalid if the timestamp lies in the past.
5. Use the RSA public key from the ltpa.keys file to validate the signature, using the user data as input.
6. If all prior checks succeeded, treat the "u" key from the user data as the authentic identifier of the current user.

The reason why the timestamp is included twice is probably for the purpose of legacy compatibility: older token issuers might only include the expiration timestamp in the unsigned part of the token, while older validators may only check this part. The attacks described here assumes that both the issuer and the validator are aware of signed expiration dates.

It turns out there were vulnerabilities in steps 1, 3 and 5, while the expiration fallback mechanism allowed for more convenient exploitation. The attacks are described in the three sections below.

Defeating Encryption: A Chosen-Ciphertext "Key Extraction" Attack

Note: this will be the only section containing cryptographic details. In case you are not interested in that: the bottom line is that an attacker with a valid token will be able to exploit a crypto bug in order to learn the AES encryption key and thus bypass validation step 1.

AES by itself is really just a low-level building block capable of permuting blocks of 16 bytes. In order to turn it into a full-fledged encryption scheme one needs to plug it into a mode of operation. There are several popular modes of operation available, with varying security properties. In the case of the LTPA protocol, the well-known Cipher Block Chaining (CBC) mode is used.

The working of the CBC mode of operation is shown in

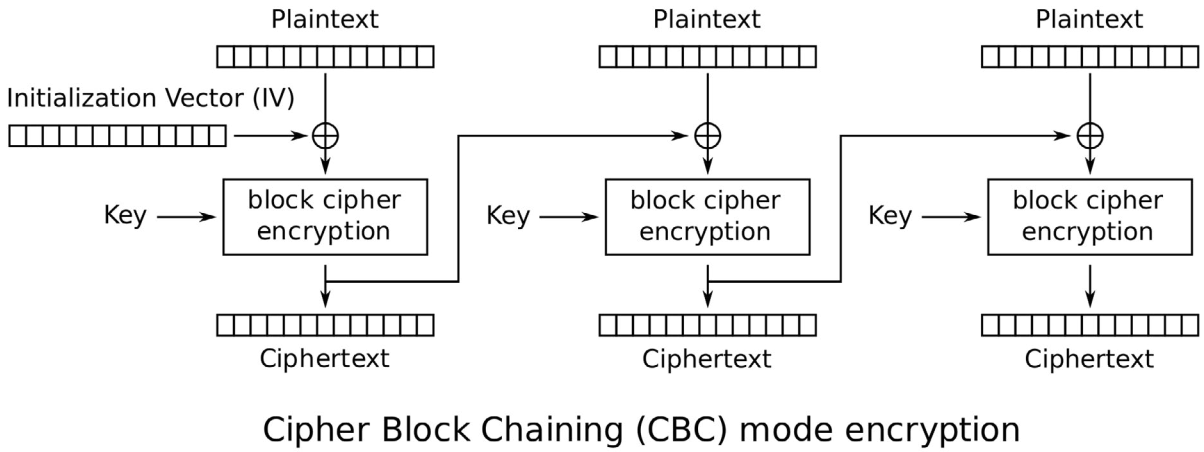


Figure 2: Diagram of the CBC mode of operation. Source: WhiteTimberwolf via Wikimedia Commons

Figure 2: before encryption, each plaintext block is XOR'ed with the value of the previous ciphertext block. The very first block is XOR'ed with an Initialization Vector (IV) value that is provided as a parameter during both encryption and decryption.

By picking CBC, the LTPA designers actually already made a (common) mistake. Even though the CBC mode is very popular, it can only offer security against chosen-plaintext attacks (an attacker can decide something to encrypt and then observe the result) but not against chosen-ciphertext attacks (an attacker can alter ciphertexts before they are decrypted). While chosen-plaintext security is sufficient in many application areas, it is definitely not enough for protecting cookies sent back and forth between a server and an untrusted user, since the user can modify cookies before sending them back to the server and directly observe its response.

Furthermore, the chosen-plaintext security property of CBC is only achieved when the IV is randomly generated and unique for every encrypted message. This requirement is however violated by the LTPA protocol, which uses the AES key as the IV value. What this means in practice is that the first 16-byte block of the ciphertext will be defined as AES encryption of the AES key XOR the first plaintext block; which is a very useful property for attackers. Because any information we know about the start of the plaintext directly tells us about the value of (parts of) the key.

A well known chosen-ciphertext attack against AES-CBC is **Vaudenay's padding oracle attack**. This attack relies on an attacker making selective alterations to a submitted ciphertext, and observing differences between two types of decryption errors in order to extract information about the plaintext. Unfortunately, this attack does not directly work against WebSphere Liberty, because servers do not disclose any details about decryption errors to the end user. In fact, the only bit of information that the server will disclose is whether token validation succeeded or not, through the HTTP status code.

This may not seem like much, but this single bit of information is actually enough to reveal parts of the plaintext. What really helps us is that the unsigned timestamp data is completely ignored when a signed timestamp is present. The server will not even validate if it contains a valid ASCII digit sequence: almost any byte sequence is allowed in this location. So, if we inject an extra ciphertext block somewhere in the middle of the unsigned timestamp, we can keep the token valid. Figure 3 shows what happens when we do this.

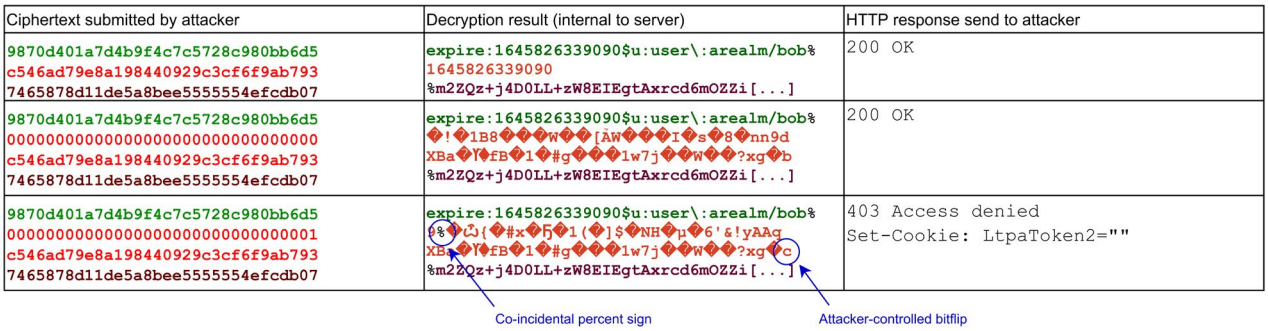


Figure 3: Effect of inserting an extra (all-zero) block in a ciphertext. Note that changing this block completely randomizes one plaintext block but causes a predictable change in another.

Figure 3 shows two effects that can happen when inserting a ciphertext block within the timestamp: either the result will contain garbage data that does not invalidate the token, or it will contain one or more bytes that happen to match the ASCII value of a percent sign. In the latter case, step 2 of token validation fails and our token is no longer accepted.

So, when we submit a ciphertext block that keeps the token valid it tells us that the result does not contain any extra percent signs. When it does become invalid, this tells us that one of the following two situations has occurred:

1. The randomized plaintext block directly corresponding to our ciphertext block happens to contain one or more percent signs (probability: about 6%).
2. When the next plaintext block is XOR'ed with our ciphertext block, the result will contain a percent sign.

In the second situation, we learn some information about the plaintext: if we observe that setting a particular ciphertext byte at position I to a value X will always result in an error, we can conclude that the plaintext byte at position I + 16 must therefore have the value X XOR '%'. This fact can be exploited to slowly retrieve the contents of a plaintext block, by using a script that implements the algorithm illustrated in Figure 4.

The trick here is that we insert a copy of the first block of ciphertext, that, upon decryption, will have a value equaling the IV (i.e. the AES key!) XOR'red with the plaintext of the first byte, XOR'red with our randomly inserted 3rd block. This modification increases the cookie length, but because the insertions are in the (unchecked) timestamp field, the constructed cookie will be treated as "okay to decrypt". Since we know what we inserted, and we know that the first block consists of the text "expire:" followed by several numbers that form a timestamp, it follows that if we can figure out exactly what the copy of block 1 decrypted to, we can derive the IV, which is also the AES encryption key itself!

Now, finding out what the copy of block 1 decrypts to, can be deduced by the oracle that will tell us if there are too many percent-signs in the decrypted value (see above). We do this by modifying the last byte of the random part of our insert after a successful validation. If this invalidates the token, an extra "%"-sign has appeared somewhere. By then flipping bits in consecutive bytes, we can determine what position the "%"-sign has appeared at, and easily calculate that byte of the key.

Figure 4: Illustration of an adaptive chosen-ciphertext attack algorithm that decrypts a single byte from the first plaintext block XOR'ed with the key. By repeating this 16 times, the entire value of P1 XOR key can be determined.

By following the steps in Figure 4, we can eventually determine the full value of the first plaintext block XOR'ed with the key. We know this plaintext block contains the term "expire:" followed by the first 9 digits of the token's expiration time. Even when we don't know when this expiration date is supposed to be exactly, it is probably safe to assume it will expire within the next 10 days. If so, we can easily guess the final digits with a brute-force attack, as shown in Figure 5.

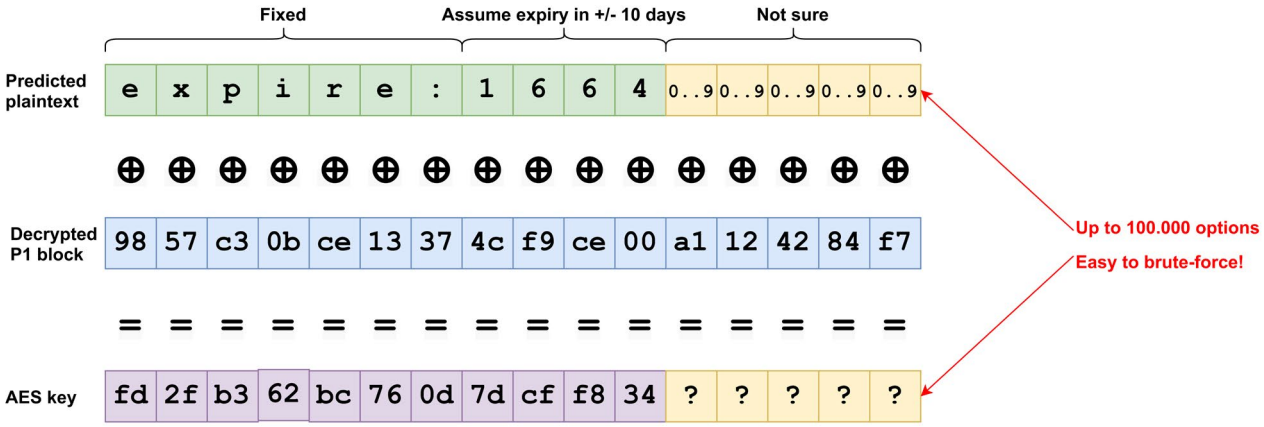


Figure 5: Separating the encryption key from the first plaintext block.

I have shared an exploit script with IBM that, given a valid LTPA token and URL of any endpoint that validates it, executes this attack and returns the AES encryption key.

With this key, it is possible to decrypt and re-encrypt tokens; thus completely negating the encryption layer of the LTPA protocol. By itself, however, this is not such an interesting capability, because it just allows us to figure out when the token expires and what the user identifier is. This is a fundamental protocol bug, though, and this attack (or some variation thereof) will likely work against other LTPA implementations as well.

I haven't been able to find an application that actually relies on LTPA tokens to keep information secret, although I can't that I have managed to successfully investigate all (proprietary and closed-source) LTPA implementations. Therefore I advised IBM to look into this themselves, and to avoid every having to rely on LTPA token secrecy.
If we want to actually tamper with token plaintexts, we also need to bypass the signature. This is where the implementation vulnerability specific to Websphere Liberty comes in.

Bypassing the Signature Check

Signature validation was implemented using the Java method shown in Figure 6. This method first calls the expiration checking method (which throws an exception when the expiration date lies in the past) and then calls the verify method (which returns whether the RSA signature is correct, or can throw an exception in case of a syntax error).

```
public final boolean isValid() throws InvalidTokenException, TokenExpiredException {
    boolean verified = false;

    validateExpiration();

    try {
        verified = verify();
    } catch (Exception e) {
        verified = false;
        throw new InvalidTokenException(e.getMessage(), e);
    }

    if (!verified) {
        if (TraceComponent.isAnyTracingEnabled() && tc.isDebugEnabled()) {
            Tr.debug(this, tc, "Invalid signature of the token " + this);
        }
    }

    return verified;
}
```

*Figure 6: LTPA signature verification method (before patch). Source: **OpenLiberty source code via GitHub**.*

What is notable about this method is that it can fail in two ways: it can either throw an exception or return false. This creates some potentially dangerous ambiguity: when callers of the method only account for the exception failure path, they may forget to also check the return value of the method.

It turns out this is exactly what happened: isValid was called from two locations within the LTPA library and in both cases the exceptions were handled but the return value was effectively ignored. This means that the actual validity of the signature is simply never checked. This is actually not that surprising. The code path where false is returned will probably never occur under normal circumstances: it is not hit when a token expires and when someone blindly tampers with an encrypted token this will most likely result in syntax error that causes an exception. This situation probably made it difficult to catch this bug during testing and explains why it had never been encountered by accident.

By abusing the weaknesses of the encryption layer, however, we can easily exploit this bug as follows:

1. Obtain a valid LtpaToken2 cookie for an attacker account.
2. Use the adaptive chosen-ciphertext attack described in the previous section to determine the encryption key.
3. Decrypt your cookie with it and change the user identifier to that of another user you want to impersonate (you can also extend the expiration date to far in the future), but leave the signature the same.
4. Re-encrypt the altered cookie and submit it. The isValid method will return false during signature validation but because decryption and parsing succeeded the token will still be accepted. You have now impersonated another user account.

I managed to successfully carry out this attack against an application running on WebSphere Liberty that accepted LTPA authentication. This vulnerability has been assigned **CVE-2022-22475**.

Do note that this attack is authenticated: you need to first obtain a valid token yourself. So one could consider this impersonation vulnerability to be closer to a privilege escalation attack than to a full authentication bypass. Of course any user being able to impersonate all others could still be very severe for multi-user web applications.

I have not been able to find a way to exploit the key extraction attack without a sample of a valid token to start with, nor have I found a practical method to use chosen-ciphertext attacks to forge a syntactically correct token from scratch. I do not want to rule out that such attacks are possible, though, which would in theory turn this vulnerability into a full authentication bypass.

Attacking the Parser: LTPA Delimiter Injection

Besides the issues with decryption and signature validation, I have also found a vulnerability in the user data parser that is exploitable without touching any cryptography.

While in practice tokens usually just contain "u" and "expire" attributes, the parser actually supports an extensive language for key-value mappings, where multiple values can be mapped to the same key. This language also allows escaping of special characters. For example, consider the following mapping:

```
"akey" => ["value1", "value2", "special character: $"]
"otherkey" => ["other value"]
```

This will be represented with the following LTPA token syntax:

```
akey:value1|value2|special character\: \$$otherkey:other value
```

Note that colons are used to separate keys and values, dollar signs are used to separate key-value pairs and pipe characters separate multiple values mapped to the same key. Backslashes are used to escape special characters when they are part of a value string.

It turned out, however, that the escaping logic of the token builder was incomplete: as shown by Figure 7, the special characters %, $ and : were escaped but the characters | and \ were not.

```
private static final char TOKEN_DELIM = '%';
private static final char USER_DATA_DELIM = '$';
private static final char USER_ATTRIB_DELIM = ':';
private static final String STRING_ATTRIB_DELIM = "|";
```

```
private static final String escape(String str) {
    if (str == null) {
        return null;
    }

    StringBuilder sb = new StringBuilder(str.length() * 2);
    int len = str.length();
    for (int i = 0; i < len; i++) {
        char c = str.charAt(i);
        switch (c) {
            case TOKEN_DELIM:
            case USER_DATA_DELIM:
            case USER_ATTRIB_DELIM:
                sb.append('\\');
                break;
            default:
                break;
        }
        sb.append(c);
    }
    return sb.toString();
}
```

*Figure 7: Character escaping logic in LTPA token builder (before patch). Source: **OpenLiberty source code via GitHub**.*

This oversight is exploitable in situations where attackers can influence or select their own user ID's. For example, when they can register on a site with a custom username or e-mail address. Consider the case where an attacker wants to impersonate the user named "admin". What they can do is register a user with a name like "admin|notreally". This will result in them being handed a token that embeds the following key value pairs:

```
"expire" => ["1234567890000"]
"u" => ["user:somerealm/admin|notreally"]
```

When included in the LTPA token, this is serialized as follows:

```
expire:1234567890000$u:user\:somerealm/
admin|notreally
```

This userdata string is then included in a properly signed and encrypted token.

At first, a WebSphere Liberty server will cache the original key-value pairs, and not bother to decrypt and parse the token again when it sees it in a request. However, there are plenty of options to force a cache miss. I made a simple ciphertext change that didn't invalidate the token, which worked right away. I can imagine that messing with load balancer cookies or waiting for the cache to be purged will also be effective. You could also send the token to a different application server than the one that issued it, considering that having multiple servers is the main reason to use LTPA in the first place.

In such cases the token will be parsed again into the following key-value mapping:

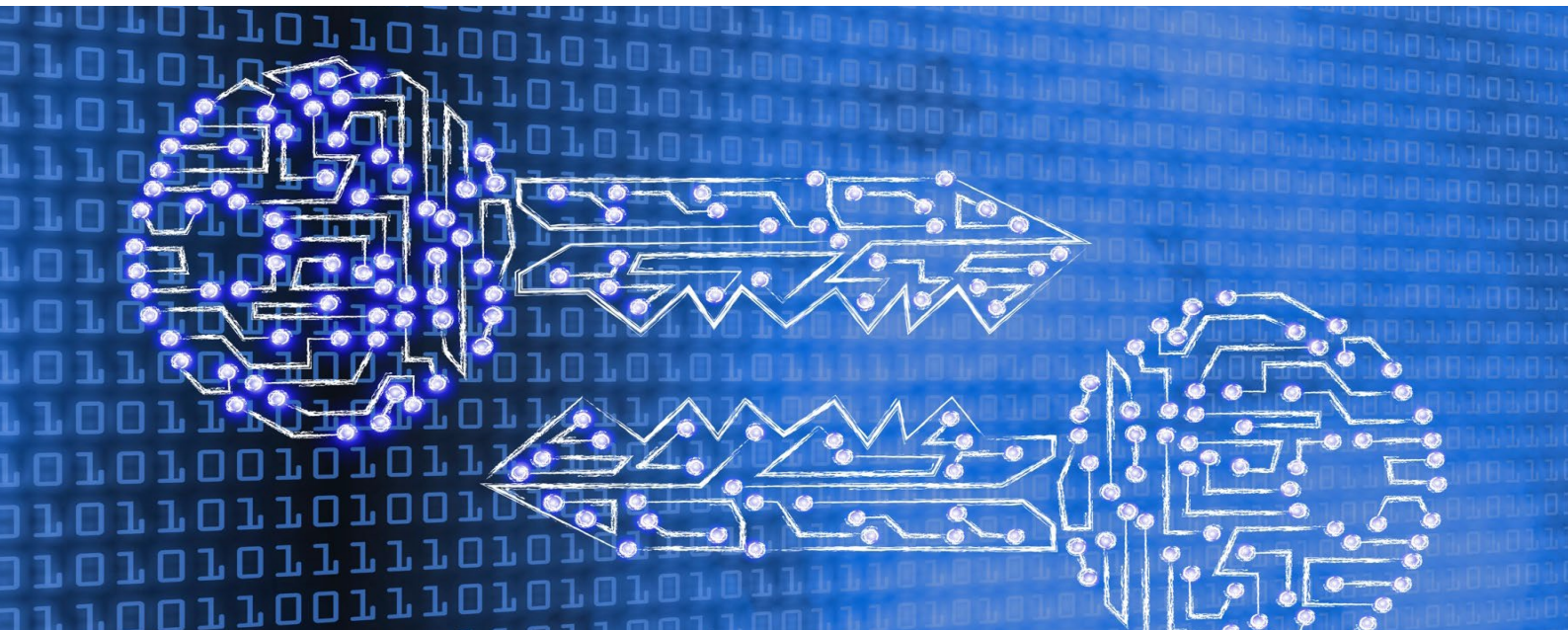
```
"expire" => ["1234567890000"]
"u" => ["user:somerealm/admin", "notreally"]
```

The user now appears to have two usernames. The implementation will however simply pick the first one, and now the attacker has logged in as "admin".

This vulnerability has been assigned **CVE-2022-22476**.
I guess that exploitable applications may be pretty rare though, as they need to meet the following criteria in order to be vulnerable:

1. The application uses an unpatched WebSphere Liberty version with LTPA authentication.
2. Users can register (or change) their own user identifiers.
3. These user identifiers are allowed to contain a "|" character.

I have not yet seen an application that meets all three criteria, and an application that does will also be vulnerable to the cryptographic attacks which do not require an account to be registered. However, there is the theoretical notion that such applications exist. Nonetheless, when an application does happen to be vulnerable, exploitation is very easy and does not require the attacker to use any automated scripts.



Recommendations

The three vulnerabilities have been responsibly disclosed to IBM, which have resulted in patches published in July 2022. If you have an application that uses WebSphere Liberty or Open Liberty and (might) use LTPA authentication, we recommend you install these patches right away. See the **IBM support pages** for more information.

Exploitation attempts of both vulnerabilities should leave a trace: CVE-2022-22475 exploitation will involve sequences of hundreds of requests containing invalid tokens, while exploiting CVE-2022-22476 requires the creation of a user account with a "|" character in the name.

Because CVE-2022-22475 has a modest CVSS score of 7.1 and has received not much publicity so far, there is a chance that installation of this patch has not received high priority by maintainers. Therefore, we have decided to not publicly release our exploit script at this time. While the creation of this script was not trivial, running it against a WebSphere server is easy and does not require cryptographic knowledge. Therefore, take into account that (easy to use) public exploits may appear in the future.

The patches mitigate the parser flaws by properly escaping all special characters, and solve the signature bypass by making the isValid method always throw an exception when validation fails. Exploitation of the encryption key extraction attack is made significantly more difficult because the unsigned timestamp is now checked to match the signed timestamp. Even if that attack is still possible, it would only allow an attacker to compromise confidentiality of a token (which is probably not useful) and not its integrity.

Note that we have not fully tested other (IBM) products such as "traditional Websphere" for a similar signature validation bugs. I did take a glance at a few open source and unofficial LTPA token validators on GitHub, and did not spot the same signature validation bug.

Because we can't rule out that the same vulnerable Liberty code base may have been used elsewhere, we recommended to IBM to investigate this. As far as we know patches have only been released for the Liberty product.

In general, I would recommend against using LTPA tokens for new applications: the underlying cryptography does not follow best practices and these attacks have shown that the complexity of the protocol is sensitive to implementation errors.

Just like Zerologon, these protocol and implementation vulnerabilities have remained undetected for more than a decade. This shows that even if an authentication protocol has been in use for a long time, that does not mean its underlying cryptography is safe or has ever been properly scrutinized.

About Secura

Secura has worked in information security and privacy for over two decades. This is why we uniquely understand the challenges that you face like no one else and would be delighted to help you address your information security matters efficiently and thoroughly. We work in the areas of people, processes and technology. For our customers we offer a range of security testing services varying in depth and scope.

Secura has the mission to support organizations with up-to-date knowledge to work toward a bright and safe future.

Stay up to date with the latest insights on digital security and subscribe to our periodical newsletter: secura.com/subscribe.

Follow us on:

Secura
A BUREAU VERITAS COMPANY

BUREAU VERITAS
Shaping a World of Trust