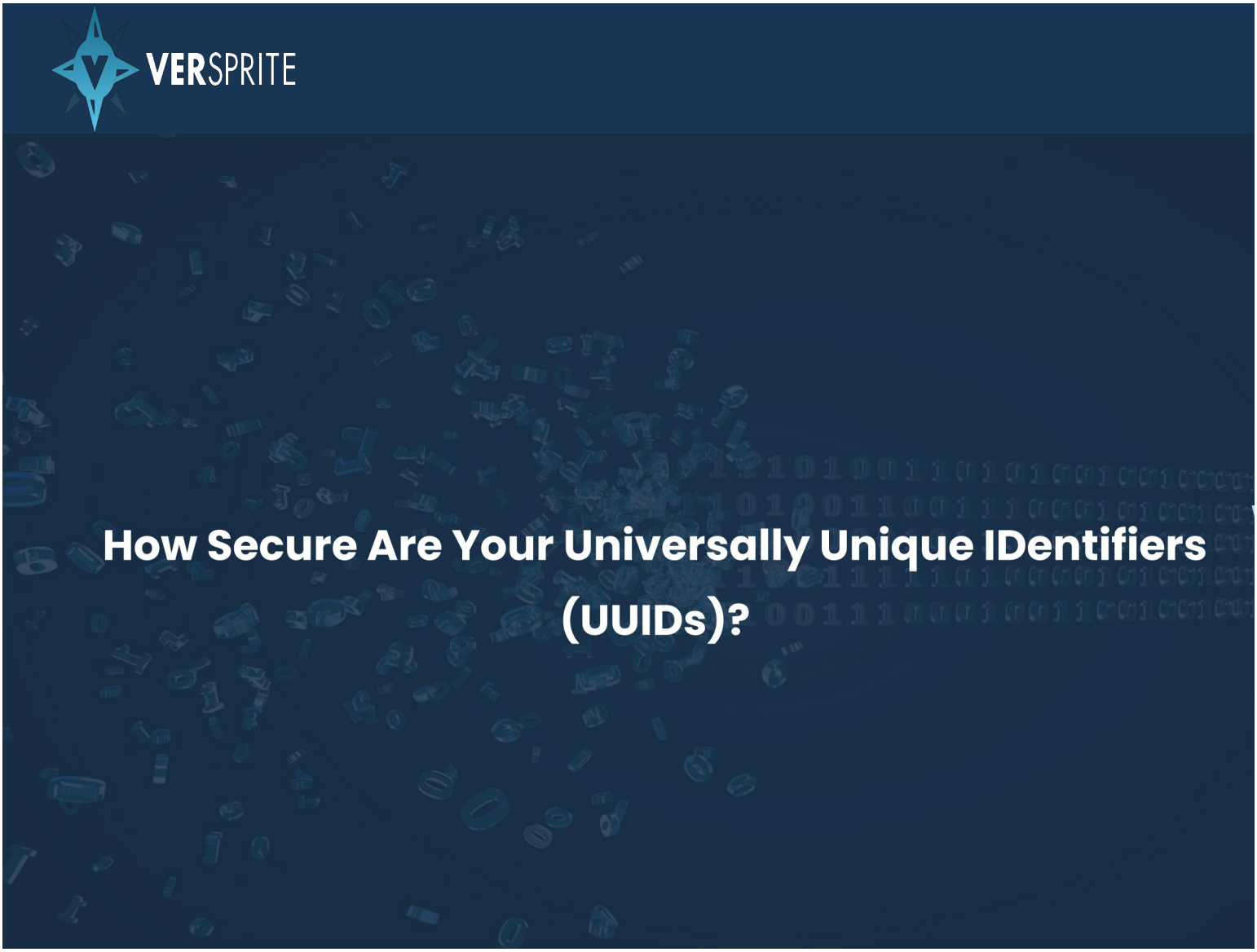

< Back to Blog Home

Inside Universally Unique Identifiers (UUIDs)

Universally Unique IDentifiers, also known as UUIDs or GUIDs, are 128-bit numbers used to uniquely identify information in computer systems. UUIDs can be used to refer a wide variety of elements (documents, objects, sessions, tokens, entities, and so on). They can also be used as database keys.

As the name implies, UUIDs should be for practical purposes unique and ideally hard to guess; although in certain scenarios – some of them which will be later discussed in this post – an attacker in possession of UUIDs that were previously generated by a system might be able to predict future ones.

UUID Format

The 16 octets of a UUID are represented as 32 hexadecimal digits and separated by hyphens into five groups, in the form 8-4-4-4-12. This results in a total of 32 alphanumeric characters and 4 hyphens. See the UUID v1 example below:

```
123e4567-e89b-12d3-a456-426655440000
xxxxxxxx-xxxx-Mxxx-Nxxx-xxxxxxxxxxxx
```

In this example:

- **M**: Indicates the UUID version. In the example above, it's UUID v**1**.
- **N**: The **1–3** most significant bits of digit, **N,** indicate the UUID variant. In the example above, **N** is **a** (1010_2), meaning that the UUID is a variant-1. From the RFC:

Variant	Msb0	Msb1	Msb2	Msb3	Description
0	0	X	X	X	Reserved, NCS backward compatibility.
1	**1**	**0**	**X**	**X**	**The variant specified in this document.**
2	1	1	0	X	Reserved, Microsoft Corporation backward compatibility.
3	1	1	1	X	Reserved for future definition.

Nowadays most implementations adopt variant-1 where the Most Significant Bit (**Msb0**) is going to be set (**1**), whereas **Msb1** is going to be unset (**0**). This leaves **N** with two "free" bits (**Msb2** and **Msb3**), meaning that it can only be one of the following: **8**, **9**, **a** or **b**.

UUID Versions: From Nil UUID to UUID v4

Nil UUID

In the UUID 00000000-0000-0000-0000-000000000000, all bits are set to zero.

UUID v1

UUIDs v1, also known as host or time based UUIDs, are generated taking into consideration different components. Take for example, the following UUID v1:

e034b584–7d89-11e9–9669–1aecf481a97b

- **Timestamp**: A 60-bit value, representing the number of 100 nanosecond intervals since 15 October 1582 00:00:00.00. In this example, timestamp has the hexadecimal value of **1e97d89e034b584**. In order to extract the time from said representation, the following calculations should be performed:

  1. Convert the hex value **1e97d89e034b584** to a decimal representation:
     - **0x1e97d89e034b584** = **137779294737053060**.

  2. Subtract **122192928000000000** (the interval between Julian and Gregorian calendar in 100 nanoseconds) to the previous number and divide by 10000
     - (**137779294737053060** – **122192928000000000**) / 10000 = **1558636673705**.

  3. We now have the time in regular epoch/Unix timestamp. Convert to date and we obtain:
     - **Thu May 23 13:37:53 CDT 2019.**

- **Version:** The UUIDs version in this example can be found in the grey text area "1", which makes the example's version UUID v1.
- **Clock Sequence / Clock ID:** A 14-bit value, originally initialized to a random value to minimize the correlation across systems. (i.e., only performed once in the lifetime of a system). In this example, it's 1669. In the UUID itself, the first digit isn't a **1** but a **9** because the most significant bit is also set, as mandated by the variant.
- **Node ID:** 48-bit MAC address of the "node" (that is, the computer generating the UUID). In this example, its **1a:ec:f4:81:a9:7b.**

Pros of UUID v1:

- You are guaranteed to get unique UUIDs every time they are generated.

Cons of UUID v1:

- An attacker may be able to get the exact **timestamp**, **clock sequence** and **node** (MAC address) of the system from an arbitrary UUID – **Cost of anonymity**.
- If an attacker is in possession of previous UUIDs generated from a system, it's much easier to **predict future ones**.
- **Never rely on UUID v1 for authorization purposes** – use UUID v4.

UUID v2

Version-2 UUIDs are similar to version 1, except the least significant 8 bits of the clock sequence are replaced by a "local domain" number, and the least significant 32 bits of the timestamp are replaced by an integer identifier meaningful within the specified local domain, however, many UUID implementations omit version 2.

UUID v3 and UUID v5

Both UUID v3 and UUID v5 are generated using the hash of namespace and name. The key difference between versions three and five is that UUID v3 uses MD5 as the hashing algorithm, whereas UUID v5 uses SHA-1.

```
UUID = hash(NAMESPACE_IDENTIFIER + NAME)
```

UUID v4

UUIDs v4 are generated almost entirely random. Only four (4) bits are used to indicate the

version and two (2) bits are used to indicate the variant (when variant-1 is adopted). The rest of the bits (122) are randomly generated. See the UUID v4 example below:

```
00e8da9b-9ae8-4bdd-af76-af89bed2262f
```

- **Version**: UUID v**4**.
- **Variant**: Instead of the **a**, also an **8**, **9** or **b** could be present in that position.

One of the disadvantages of using UUID v4 is that there is a small chance of generating duplicated UUIDs. Although, with the high number of possible combinations (2^122), this situation is highly unlikely to happen.

Pros of UUID v4:

- UUIDs v4 is generated **randomly**.
- They are almost 100% anonymous.

Cons of UUID v4:

- There is a chance that a **UUID could be duplicated**. However, the probability is low.
- Sometimes secure cryptographic functions are not used, or used incorrectly, giving attackers the chance to compute future outputs of the PRNG that will produce the same UUIDs – Check out this **tool** and this blog post: **Cautionary note: UUIDs generally do not meet security requirements**.

Sandwich Attack: A New Way Of Brute Forcing UUIDs

Consider the following scenario: A web application allows users to reset their password by means of a “Forgot Password” functionality. After following the password reset process, a new UUID v1 is generated and an email with a unique reset link is sent to the email associated with said user. The reset link may look something like this:

```
https://www.acme.com/reset/836d28b2-7592-11e9-8201-bb2f15014a14
```

Once an attacker knows the web application uses UUID v1 for generating the password reset link, they could take the approach listed below to guess the right token for an arbitrary account:

1. Send a password reset request for an account the attacker controls, for example, attacker1@acme.com.
2. Immediately after, send a password reset request for the targeted account (victim@acme.com) followed by a request for another account owned by the attacker (attacker2@acme.com). If multiple password reset requests are allowed in a short period, the first account can be used again (attacker1@acme.com). In order to carry out the attack as quickly as possible, tools like **Turbo Intruder** (Burp Suite extension) or **racepwn** can be used.
3. Compare the URLs obtained for each password reset request:

```
https://www.acme.com/reset/99874128-7592-11e9-8201-bb2f15014a14
```

```
https://www.acme.com/reset/998796b4-7592-11e9-8201-bb2f15014a14
```

4. Brute force the hexadecimal range between **998796b4** and **99874128** until the victim's reset link is found. In this particular example, the space is considerably short (~**22,000** combinations). If no rate limiting protections are in place, the link can be found in less than an hour using standard computing resources.
5. The targeted account's password reset link will be something in the form:

```
https://www.acme.com/reset/99876914-7592-11e9-8201-bb2f15014a14
```

A possibility exists where multiple back- end servers are present behind a single

application, or that load balancing is taking place. In this scenario, the attack can be significantly harder to carry out, as UUIDs are going to vary from one server to another. If this is the case, an attacker must try to find a way of hitting the same server consistently (i.e. if the domain name for the targeted application resolves to different public IPs, modify your local 'hosts' file so the domain always resolves to the same IP) or perform the attack multiple times until the same server is hit for all requests.

Another challenge to carrying out the attack in this example is that, if there is a significant delay between requests, the first 4 octets of the UUIDs are going to vary considerably from each other making it impractical (the hexadecimal range to brute-force would be too big).

Faulty Pseudo-Random Number Generators (PRNGs) with UUIDs v4

As stated by Scott Contini in "Cautionary Note: UUIDs generally do not meet security requirements", sometimes UUID v4 is used, but the pseudo-random number generator (PRNG) is faulty.

Take a look at the function that was used for generating 'random' UUIDs:

```
function randomUUID() { var s = [], itoh = '0123456789ABCDEF'; // Make array of random hex digits. The UUID only has 32 digits in it, but we // allocate an extra item to make room for the '-'s we'll be inserting. for (var i = 0; i < 36; i++) s[i] = Math.floor(Math.random()*0x10); // Conform to RFC-4122, section 4.4 s[14] = 4; // Set 4 high bits of time_high field to version s[19] = (s[19] & 0x3) | 0x8; // Specify 2 high bits of clock sequence // Convert to hex chars for (var i = 0; i < 36; i++) s[i] = itoh[s[i]]; // Insert '-'s s[8] = s[13] = s[18] = s[23] = '-'; return s.join(''); }
```

In this example, the **randomUUID()** function uses **Math.random()** which by itself is not cryptographically secure. Contini went further into actually understanding how the Math.random() was implemented in JavaScript, and developed a script to brute-force the **hi** value given a random UUID generated with the vulnerable function in order to predict future outputs (the script can be downloaded from **here**).

Examples like this show us that every time we are able to obtain the source code responsible for generating UUIDs in a web application, it is worth taking a look at how that is implemented. Sometimes developers don't use secure functions or even develop their own, which may lead to cryptographic vulnerabilities (Contini, 2015).

Security Considerations When Building an Application

There are several security considerations for each version of UUIDs to consider when building an application. The best solution is to go with the most secure version of UUIDs for the kind and nature of the application you're building.

For automated passive testing use the **UUID issues for Burp Suite** extension (also available in PortSwigger repository).

Web Application Security

VerSprite focuses on emulating cybercrime and simulating test scenarios that not only reflect current attack patterns, but also threat motives. The foundation of VerSprite's penetration testing methodology is based on emulating realistic attacks by a malicious actor through the use of PASTA (Process for Attack Simulation and Threat Analysis). **Read More →**

Penetration Testing Methodology: Emulating Realistic Attacks by a Malicious Actor