Till REcollapse

Till REcollapse - 0xacb, 0xacb.com.

- Published: date not stated
- Original: <https://0xacb.com/2022/11/21/recollapse/>
- Preserved from: https://0xacb.com/2022/11/21/recollapse/ (live) on 2026-08-10
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

Till REcollapse - 0xacb

Welcome back to my blog. In this post, I'll explain the REcollapse technique. I've been researching it for the last couple of years to discover weirdly simple but impactful vulnerabilities in hardened targets while doing bug bounties and participating in HackerOne LHEs. This technique can be used to perform zero-interaction account takeovers, uncover new bypasses for web application firewalls, and more.

This post is mostly based on my BSidesLisbon 2022 talk and follows the launch of the recollapse tool, now available on GitHub. This is also something we started researching internally at Ethiack.

Dealing with user input

It all starts with unexpected input. Modern applications and APIs rely on validation, sanitization, and normalization. This is usually done by custom regular expressions and widely used libraries that validate and transform typical user input formats, such as email addresses, URLs, and more.

Here are some validation and sanitization examples:

```
>>> re.match(r"^\S+@\S+\.\S+$", "aa.com")
>>> re.match(r"^\S+@\S+\.\S+$", "[[email protected]](https://0xacb.com/cdn-cgi/l/email-protection)")
<re.Match object; span=(0, 7), match='[[email protected]](https://0xacb.com/cdn-cgi/l/email-protection)'>
```

Validation (Python)

```
> htmlspecialchars("input'\"><script>alert(1);</script>");
= "input'"><script>alert(1);</script>"
```

Sanitization (PHP)

The goal is always about preventing dangerous user input from being stored in the first place. Let's consider an application that rejects special characters in the `name` of a user on a `/signup` endpoint. An attacker can't inject payloads in the `name` but this doesn't necessarily mean that, later on, the `name` would not be sanitized somewhere, resulting in vulnerabilities, such as XSS. In this case, we can try to find alternative endpoints that are more permissive and accept special characters in the same parameter. This is what I did with @itscachemoney back in 2019 in Dropbox, or... we can try to find a bypass for the regex in a black-box manner like I'll show later in this post.

In the other hand, normalization is used to make user input consistent. It's handy for applications with multiple account flows to avoid duplicate email addresses, such as `[[email protected]](https://0xacb.com/cdn-cgi/l/email-protection)` vs `[[email protected]](https://0xacb.com/cdn-cgi/l/email-protection)` vs `á@ª.com` and so on. The normalization libraries have different outputs, as you can see in these examples, which can be helpful to detect technologies used by the backend.

```
> iconv("UTF-8", "ASCII//TRANSLIT", "Ãéï°úç");
= "~A'e\"i^0'uc"
```

Normalization (PHP)

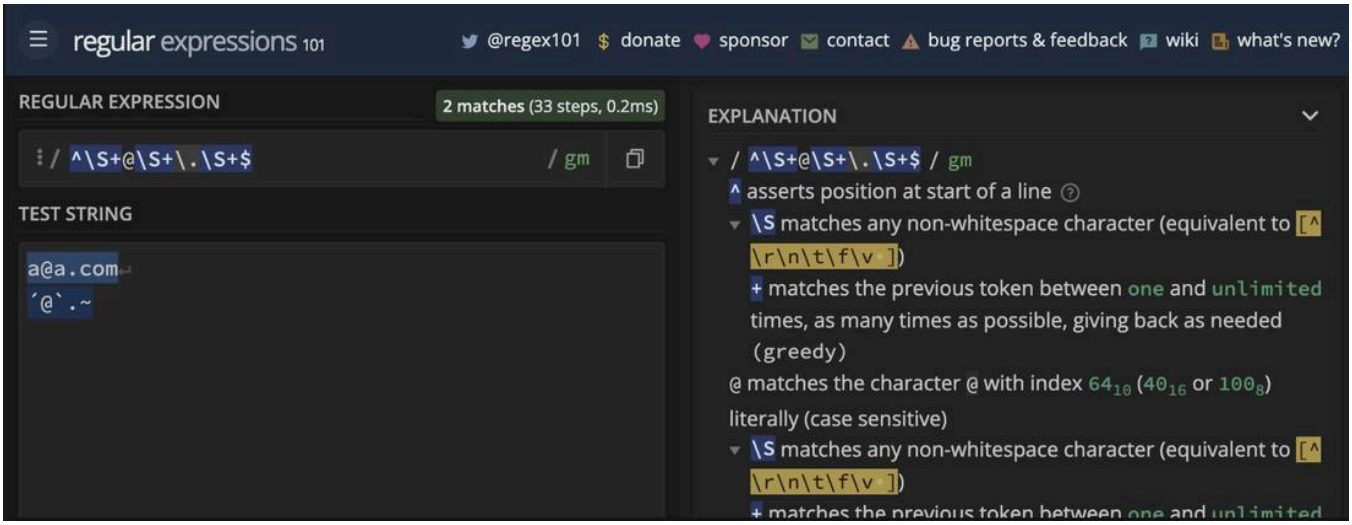
```
>>> unidecode.unidecode("Ãéï°úç")
'Aeideguc'
```

Normalization (Python)

You can find more information on normalization if you are unfamiliar with it here: https:// s . / /. However, browser normalization behavior is just the tip of the iceberg.

What's the problem?

Regex is usually reused from StackOverflow, Github, or other sources. Developers typically don't test them properly and sometimes paste different regular expressions across backend endpoints. For instance, the aforementioned regex `"^\S+@\S+\.\S+$"` doesn't work well for proper email validation:



regex101.com

For mature targets, testing code exists but can be specific to a subset of the possible cases. In the following scenario, injecting quotes still goes through the assertion:

```
>>> msg = 'Entity "test" is not available'
>>> assert re.match(r'^Entity ".+" is not available$', msg)
>>> msg = 'Entity ""><h1>x" is not available'
>>> assert re.match(r'^Entity ".+" is not available$', msg)
```

Example of testing code

Things also get interesting with GitHub Copilot. Generating code to validate if an URL is part of a whitelisted domain gives the following result in Python:

```
def url_is_subdomain(url, domain):
    """Return True if url is a subdomain of domain."""
    return re.match(r'^(?:https?://)?(?:[^/]+\.)?%s(?:/.*)?$' % domain, url)
```

Code generation with Copilot

Fuzzing this regex with the REcollapse tool presented bellow gives an input `https://example com` that will be accepted for `example.com` as the domain argument, but it's translated to `xn--examplecom-ehl` (punycode), allowing an attacker to bypass the validation.

Why could normalization be a problem?

In terms of normalization, confusion and duplicate states can sometimes be reached if normalization is not used consistently in all endpoints and flows. Let's say we have a victim with the email `[[email protected]](https://0xacb.com/cdn-cgi/l/email-protection)`. An attacker may try to explore all flows with the email `hil°[[email protected]](https://0xacb.com/cdn-cgi/l/email-protection)`.

```
>>> unidecode.unidecode("hil°[[email protected]](https://0xacb.com/cdn-cgi/l/email-protection)")
'[[email protected]](https://0xacb.com/cdn-cgi/l/email-protection)'
>>> unidecode.unidecode("victim@exámple.com")
'[[email protected]](https://0xacb.com/cdn-cgi/l/email-protection)'
```

Normalization (Python)

This also applies to the domain part, which can result in being able to receive a recovery link on a punycode domain for `[[email protected]](https://0xacb.com/cdn-cgi/l/email-protection)` on `victim@exámple.com`, which resolves to `[[email protected]](https://0xacb.com/cdn-cgi/l/email-protection)`, potentially resulting in zero-interaction ATO.

This can also be applied to SSO or OAuth flows if the source or the destination app normalizes critical identifiers, such as email addresses.

We are not the same

The core regex libraries of different programming languages can have slight differences while processing the same regular expression. Let's consider the following common description of the dollar sign:

>

$ asserts position at the end of the string, or before the line terminator right at the end of the string (if any)

```
> "aaa".match(/^[a-z]+$/)
[ 'aaa', index: 0, input: 'aaa', groups: undefined ]
> "aaa123".match(/^[a-z]+$/)
null
> "aaa\n".match(/^[a-z]+$/)
null
> "aaa\n123".match(/^[a-z]+$/)
null
```

JavaScript

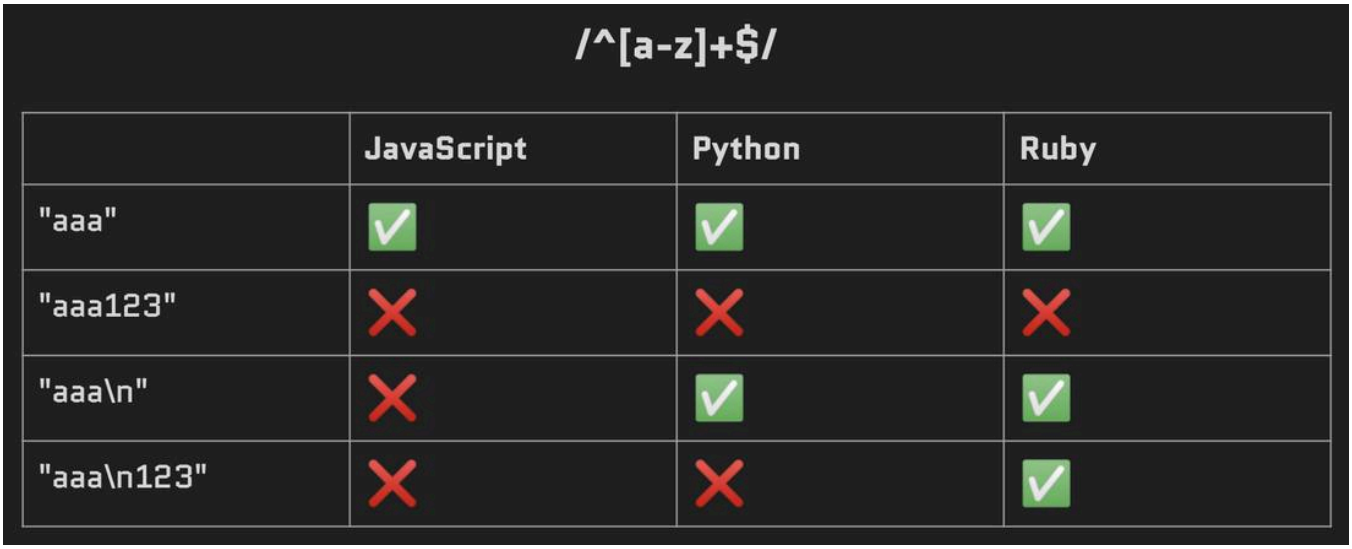
```
>>> re.match(r"^[a-z]+$", "aaa")
<re.Match object; span=(0, 3), match='aaa'>

>>> re.match(r"^[a-z]+$", "aaa123")
>>> re.match(r"^[a-z]+$", "aaa\n")
<re.Match object; span=(0, 3), match='aaa'>

>>> re.match(r"^[a-z]+$", "aaa\n123")
```

Python

```
irb(main):001:0> "aaa".match(/^[a-z]+$/)
=> #<MatchData "aaa">
irb(main):002:0> "aaa123".match(/^[a-z]+$/)
=> nil
irb(main):003:0> "aaa\n".match(/^[a-z]+$/)
=> #<MatchData "aaa">
irb(main):004:0> "aaa\n123".match(/^[a-z]+$/)
=> #<MatchData "aaa">
```

Ruby

Testing the same regex and input pairs on different libraries without setting multiline regex flags, leads to different behaviors:



Comparison table

Another problem is that developers usually validate the input and still use the original one instead of extracting the matching part. Using `^` and `$` to assert the beginning and end of the string can still allow newline characters, or these assertions can be missing.

The REcollapse technique

So, how to bypass the current validation or sanitization? Also, how can we leverage user input transformations? **Fuzz the parameters in a smart way.**

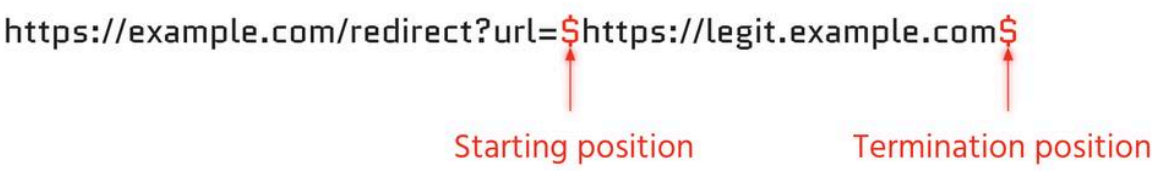
Consider the following scenario:

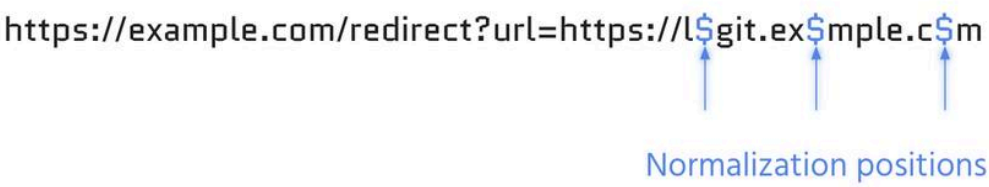
https://example.com/redirect?url=https://legit.example.com https://example.com/redirect?url=https://evil.com

We can't redirect to an attacker-controlled URL at first glance. Trying a bunch of payloads also doesn't work. What can we do?

1. Identify the regex pivot positions

- Starting & termination positions
- Beginning and end of the input
- Separator positions
- Before and after special characters
- Normalization positions
- Typically vowels `ª` > `a`





1. Fuzz positions with all possible bytes `%00` to `%ff`. Here you can see more examples:

```
https://legit.example.com → $https$:$/$/$l$git$.$ex$mple$.$c$m$
legit@example.com → $l$git$@$ex$mple$.$c$m$
user_name → $us$r$_$n$me$
<a href=x>y</a> → $<$$$ $hr$f$=$$$>$$$<$/$$$>$
```

1. Analyze the results: sort by response codes or response length.

The REcollapse tool

The REcollapse tool can generate inputs according to these rules, and supports multiple fuzzing sizes and encodings. It can also be helpful to bypass WAFs and weak vulnerability mitigations. The goal of this tool is to generate payloads for testing. Actual fuzzing shall be done with other tools like Burp (intruder), ffuf, or similar. Manual and creative work is usually still required to take any bypass to the next level.

It's available at: https://github.com/0xacb/recollapse

Resources

For bug examples, take a look at my BSidesLisbon or NahamCon slides.

Talk videos with more in-depth explanations have been published on YouTube: NahamCon 2022 EU, BSidesLisbon 2022.

A normalization table is also available here: https://0xacb.com/normalization_table

---

Takeaways

- Developers: always test and fuzz your regex, or rely on well-known libraries
- Simple input modifications can result in great damage
- Fuzz by flipping or adding bytes
- Black-box regex testing is still not very touched
- Regex behavior can reveal information about libraries and technologies
- If something is being validated and you can bypass it...
- Think about the impact, and you'll see the big picture!

Special thanks

- @regala_
- @0xz3z4d45
- @jllis
- @samwcyo
- @yassineaboukir
- @0xteknogeek
- @vgpinho
- BBAC
- @ethiack team
- @0xdisturbance team
- @hacker0x01 team

>

Until next time, 0xacb

Previous Post Next Post
