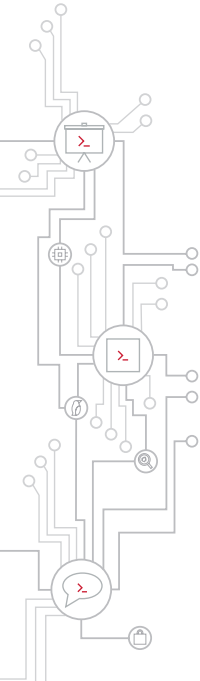

Towards More Secure Code
or
Why Devs Should Make My Job Harder

David Gstir
sigma star gmbh
2023-07-13

Hello!

David Gstir:

- Spent a lot of time writing and designing code
- Now: stares at other people's code and helps them ~~fix~~ improve it (security- and other-wise)
- Focus:
  - software security
  - Cryptography
  - Software architecture
  - Linux

sigma star gmbh

Core Areas:

- Embedded Systems
- Linux and Linux Kernel
- Cryptography

Services:

- Engineering Consulting (Systems Engineering, Security Engineering, Troubleshooting/Debugging)
- Security Consulting (Assessments, Research, Design)
- Trainings

Software Security

Insecure Software

Just some severe vulnerabilities since June 2023:

- **Microsoft:** Remote-code execution (RCE) flaw that can be exploited by maliciously crafted Microsoft Office files (CVE-2023-36884)
- **Apple:** RCE in WebKit (Apple macOS, iOS, iPadOS, CVE-2023-37450, **unpatched**)
- **Google:** Privilege escalation in Google Pixel devices (CVE-2023-21399, *details not yet announced*)
- **Linux:** Mainline Linux Kernel (v6.1+) privilege escalation (CVE-2023-3269, *StackRot*)
- **Fortinet:**
  - RCE via publicly accessible VPN interface in FortiOS (CVE-2023-33308)
  - RCD via TLS DPI feature of FortiProxy (CVE-2023-33308)
- **Barracuda:** RCE in E-Mail Security Gateway devices (CVE-2023-2868, **needs new Hardware if compromised**)
- **Cisco:** Information leak (broken cryptography) in Cisco ACI Multi-Site CloudSec on Nexus-9000 series (CVE-2023-20185, **wont fix**)
- **Mastodon:** RCE via media attachments (CVE-2023-36460)
- ...

Feels Like…


SECURITY VULNS
LIGHTYEAR
SECURITY VULNS EVERYWHERE
imgflip.com

Why is Software Insecure?

An attempt based on personal experience:

- Software is complex
- Time/budget constraints: *There is no time, we have to just make it work!*
- Requirements change
- We don't use full potential of tools
- We constantly add new dependencies, frameworks, languages and technologies
- Engineers have **too little knowledge about good practices**
- ...

How Do We Improve That?

- There is no silver bullet (sorry Rust ppl.)
- Include security at every step of project (design to maintenance)
- Identifying threats:
  - Threat modeling
  - Risk management
  - ...
- Prevention & mitigation:
  - Code reviews
  - Static and dynamic code analysis tools
  - External security audits, pentests etc.
  - Defense-in-depth
  - **Learn about potential pitfalls**
  - ...

Security Auditor PoV

- My job is hard when I cannot find vulns in your system:
  - I have to explain what is good in your system instead of listing vulnerabilities
  - Might get me to question my expert knowledge ;-)
- However, we still find simple mistakes in production systems
- These are often easy to avoid!
- **Devs: learn more about common security vulnerabilities to avoid them!**

Let’s Play a Game!

Vulnerability Examples

- Let's look at some *easy to avoid* vulnerabilities
- All examples are based on real issues, but modified for demonstration
- There is no particular order, grouping by topics, …

Play along:

- Assume you do a code review
- You get 1 point per example if you wouldn't have approved the merge request
- Be honest with yourself! ;-)

1. Privileges

Process Privileges (1/3)

Process list:

```
1 $ ps aux
2 [...]
3 root   2805  0.1  0.2  17496 10988 ?  Ss   09:52   0:00 /opt/your-app
4 [...]
```

Process Privileges (2/3)

- There are many reasons why you might **think** you need to run as `root`
- You usually don't!
  - You need to know which files your app uses!
  - `chmod 777 <all-folders>` can also be symptom of that
- Instead:
  - Just need a single capability (`man capabilities(7)`)
  - Do the setup with high privileges and drop them afterwards
  - Fork a process with high privileges and do main app logic in unprivileged process

CVE-2023-2868

- Barracuda E-Mail Gateway vuln. recently in news
- Processed mail attachments using Perl
- Processing was done with high privileges
- This allowed full compromise (rootkit installation etc.)
- See https://www.mandiant.com/resources/blog/barracuda-esg-exploited-globally

But I run everything in containers!

- `docker run -u 0`
- `docker run --privileged`
- `docker run --cap-add=ALL`
- ...
- Same is possible in Kubernetes and friends

2. HTTP Requests

Making HTTP Requests (1/2)

```
1  resp, err := http.Get(url)
2  if err != nil {
3      // handle error
4  }
5  defer resp.Body.Close()
6  body, err := io.ReadAll(resp.Body)
7  // ...
```

Making HTTP Requests (2/2)

- Assume `resp.Body` is huge and `url` is untrusted
- Attacker can use up all your RAM and cause DoS
- Instead: limit read size
- Other languages have similar API that promotes misuse

3. Url Verification

Url Verification (1/2)

```
1  const wantedDomain = "sigma-star.at"
2
3  func isValidUrl(rawUrl string) bool {
4      u, err := url.Parse(rawUrl)
5      if err != nil {
6          return false
7      }
8      if u.Scheme != "https" && u.Hostname() != "localhost" {
9          return false
10     }
11     return strings.HasSuffix(u.Hostname(), wantedDomain)
12 }
13
14 func main() {
15     // ...
16     if !isValidUrl(userInput) {
17         log.Fatalln("Invalid␣url")
18     }
19     // ...
20 }
```

Url Verification (2/2)

- Buy **`evil-sigma-star.at`** now, it's still available!
- `strings.HasSuffix()` is insufficient here
- RegEx can be nasty too:
  - DoS via RegEx in Node.JS and friends
  - `^foo.sigma-star.at$` matches `foodsigma-star.at`

4. Clearing Buffers

Clearing Buffers (1/2)

Since *Heartbleed* we all know clearing secrets from memory is important

```
1  int encrypt_it(uint8_t *buf, size_t buf_len, char *keypath)
2  {
3      int ret;
4      uint8_t *key;
5
6      key = load_key(keypath);
7      // ...
8
9      ret = encrypt(buf, buf_len, key, key_size);
10     // ...
11
12     memset(key, 0, key_size);
13     free(key);
14     return ret;
15 }
```

Clearing Buffers (2/2)

- Compilers can remove and partially re-arrange code
- Here, the optimizer in the compiler will remove `memset` as `key` is freed anyways
- This can happen with other languages too!
- Crypto libraries offer constructs which are protected against accidental removal (e.g. `OPENSSL_cleanse`, `explicit_memset`, `explicit_bzero`, `memset_s`)

5. Access Checks

Access Checks (1/2)

```
1  int serve_file(char *sanitized_path)
2  {
3      struct stat stbuf;
4      int ret, fd;
5
6      ret = stat(sanitized_path, &stbuf);
7      // ...
8
9      // only serve our own files
10     if (stbuf.st_uid != www_uid) {
11         return EPERM;
12     }
13
14     fd = open(sanitized_path, O_RDONLY);
15     // ... serve file
16
17     return 0;
18 }
```

TOCTOU

- This is a time-of-check vs. time-of-use vulnerability
- Between `stat` and open anything can happen
- Another thread can change things
- In this case we delete the file and create another one in that place
- E.g. a symlink to `/etc/shadow`
- Secure: first open, then `fstat`

6. Processing Input

Processing Input (1/2)

```
1  static int do_mgmt_client_cmd(struct child_ctx *ctx, int client_fd)
2  {
3      char buf[512] = { 0 };
4      ssize_t n;
5
6      n = read(client_fd, buf, sizeof(buf));
7      if (n > 0) {
8          // ...
9          if (strcmp("gethostname", buf) == 0) {
10             char hostname[HOST_NAME_MAX];
11             gethostname(hostname, sizeof(hostname));
12             // ...
13
14             n = snprintf(buf, sizeof(buf), "ok!\t%s\n", hostname);
15         } else { // ... }
16
17         write(client_fd, buf, n);
18     }
19     return 0;
20 }
```

Processing Input (2/2)

- Actually two bugs:
  - We assume input string in `buf` is `NULL` terminated
  - `snprintf` return code can be below `0`
- Checking return codes is crucial!
- Reading documentation often suffices
  - We've found vulns by simply reading the `man` page

7. HTTP File Upload

HTTP File Upload (1/3)

Common Task:

- File upload via HTTP endpoint
- Only images are allowed
- What can go wrong? Ideas?

HTTP File Upload (2/3)

HTML on client:

```
1 <form action="upload" method="post"
2         enctype="multipart/form-data">
3     <input type="file" name="file" />
4     <input type="submit" value="upload" />
5 </form>
```

HTTP File Upload (3/3)

POST payload:

```
1  Content-Type: multipart/form-data; boundary=----6831
2  Content-Length: 247
3
4  ----6831
5  Content-Disposition: form-data; name="file"; filename="t.txt"
6  Content-Type: text/plain
7
8  [ file-content ]
9
10 ----6831
```

Solution

Common things to get wrong:

1. Authorization: Who is authorized to upload files
2. Max. Upload size: Out of RAM or HDD space - DoS; with cloud storage: high costs
3. Encryption during transport –> HTTPS - incl. all the problems that go with SSL/TLS
4. Path sanitation: Avoid attacker from overwriting critical system files or other uploaded files
5. File type restriction: How to avoid unwanted file types (eg executables)
6. File content check

8. JSON Web Tokens

Authorization with JWT (1/2)

Setup:

- Consider a REST API
- Fronted is a SPA written in Angular, React or other fancy Framework
- You need to authenticate your users and choose to issue then a JWT after login

Problems:

- How long is a JWT valid?
- How do you revoke a JWT?
- How do you store it in your SPA?

Bonus:

- What configuration do you use for JWT?
- What properties of the JWT do you verify on every request?

Authorization with JWT (2/2)

- JWT are intended as short-lived tokens
- Ideally they are used only once
- There is no revocation for individual JWTs! You'd need to keep a list yourself
- Storing in SPA:
  - Likely accessible to your JS code –> XSS vulns hurt
  - You could use a cookie –> what is the point of JWT then?
- Most of the time a plain old session cookie would suffice
- For OAuth2/OIDC flows you will still use JWT and there it (mostly) makes sense
  - But, don't use a flow that stores the JWT in the SPA/browser again!

9. Encryption

Crypting Things (1/3)

Encrypt data:

- Use AEAD cipher AES-GCM
- `genNonce()` generates a random 12-byte nonce (*number used once*)
- `newAESGCM(...)` initializes a new `cipher.AEAD` for AES-GCM with `key`
- `aesgcm.Seal(...)` encrypts and authenticates the plaintext in-place, auth tag is append
- `aesgcm.Open(...)` verifies auth tag and decrypts in-place

Crypting Things (2/3)

```go
package main

import (
    "crypto/aes"
    "crypto/cipher"
    "encoding/hex"
    "fmt"
    "math/rand"
)

// Builds new AES-GCM cipher with given key
func newAESGCM(key []byte) (cipher.AEAD, error) {
    block, err := aes.NewCipher(key)
    if err != nil {
        return nil, err
    }

    return cipher.NewGCM(block)
}

func genNonce() ([]byte, error) {
    var nonce = make([]byte, 12)
    if _, err := rand.Read(nonce); err != nil {
        return nil, err
    }

    return nonce, nil
}
```

```go
func encryptAndAuth(pt, key []byte) (ct, nonce []byte, err
      error) {
    aesgcm, err := newAESGCM(key)
    if err != nil {
        return nil, nil, err
    }

    nonce, err = genNonce()
    if err != nil {
        return nil, nil, err
    }

    ct = aesgcm.Seal(nil, nonce, pt, nil)
    return
}

func decryptAndAuth(ct, nonce, key []byte) ([]byte, error) {
    aesgcm, err := newAESGCM(key)
    if err != nil {
        return nil, err
    }

    return aesgcm.Open(nil, nonce, ct, nil)
}
```

Crypting Things (3/3)

Problem:

- Insecure random number generation with `math/rand`
- This might generate predictable random numbers!
- AES-GCM breaks fatally in case nonce is ever reused for same key (see VPNs, TLS)
- There are recommended upper message size limits for using the same key (see NIST SP 800-38D)

Fix:

- Use `crypto/rand` for random numbers
- Ensure that limit for use of same key is not exceeded
- Random nonce is ok in some cases, but better use a counter for nonce

10. Password Hashing

Password Hashing (1/2)

Which one should I use to store passwords?

```
1  string hash_password1(string password)
2  {
3      return MD5(password);
4  }
5
6  string hash_password2(string password)
7  {
8      string salt = random_salt();
9      return SHA256(password, salt) + "." + salt;
10 }
11
12 string hash_password3(string password)
13 {
14     string salt = random_salt();
15     return PBKDF2(HMAC_SHA256, password, 60000, 256)
16 }
17
18 string hash_password4(string password)
```

Password Hashing (2/2)

- None of these is secure:
  - MD5: *lol*
  - SHA256: You can rent hardware that breaks that quite cheap
  - PBKDF2: Too low iteration count (see LastPass!)
  - bcrypt: Iterations too low
- Recommended Argon2id with recommended parameters
- If you cannot use that, `bcrypt`, `scrypt` and `PBKDF2` are okay for legacy applications
- **Adjust parameters regularly as per recommendations!**
- See OWASP cheat sheet series on password storage
  https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

Wrapping Up

Wrapping Up

How's the score?

- 0-3: oh boy…
- 4-6: getting there…
- 7-9: okay! :)
- 9+: <3

Software Security:

- Writing secure code is hard
- Every dev. contribute by being security conscious

FIN

Thank you!

Questions, Comments?

David Gstir
david@sigma-star.at