Secure Coding Practices: A General Guideline

Jay Rovacsek c3146220@uon.edu.au

October 22, 2018

Contents

1 Preface

1.1 Running with Scissors

Admittedly, the title for this section is very much thanks to one of the first items[1] I read sections of while creating this document. The analogy for development in terms of security could not be more apt for a large portion of the development community.

Why cover this topic? As a security enthusiast and developer, I often found myself looking at a system left untouched until absolutely required, the design choices, logic and knowledge of the language it was written in left with the author. Commonly a requirement for a hotfix was/is needed in a number of this circumstances as a number of critical business services and resources may rely on the system in question.

A large portion of this paper will focus around the more common exploited vectors of web applications, however the vectors commonly exploited in web application settings are commonly exploitable in a desktop application setting, this becomes more and more important to remember as large numbers of commonly used software move to enable cross platform compatibility by utilizing technologies such as Electron[8].

Security as a serious concern is only just now becoming much more "mainstream" to companies than it had previously been, movements pushing HTTPS such as Lets Encrypt[9] or high profile individuals such as Troy Hunt[7] have aided the process of mitigating some of the most easily exploited vectors such as MiTM attacks on unencrypted communications, a plethora of cybersecurity issues however still remain present in modern organisations, with the potential damage to both organisation and individual such as recent breaches in: Sony[2], Equifax[4] and a number of other recent high profile breaches of modern history.

1.2 Tripping Over

Security in programming ~~can be a~~ *is* hard beast to tame. Some languages arguably do much better in avoiding issues being caused by users new to the language or unskilled in understanding potential issues with the code they have written. We can certainly critique early languages for the level of access to the machine they allow a user, without careful consideration in design and a well founded knowledge in the language used issues notorious of early languages. However, in this day and age of highly abstracted languages and frameworks have we traded old demons for new, or do we really have more safety in our computing goals?

As suggested by Wheeler:[3] (Page 8)

> Many programmers don't intend to write insecure code - but do anyway.

2 Common Issues in all Languages

2.1 User Input Sanitisation

User input sanitisation

2.2 ReDoS

Given the requirement for string filtering is extremely common in all applications, ReDoS or Regex Denial Of Service[6] vulnerabilities are a very real threat in application security. A number of common defenses can be used against ReDoS attacks, application of which are extremely simple:

1. Atomic grouping in Regex
2. Regex lifetime limits
3. Sanitisation of input (Although this defeats the reasons to allow for Regex patterns to be used and is very easy to not implement correctly)

2.2.1 Atomic Grouping

Atomic grouping in Regex is a group that when Regex is no longer utilising the group is thrown away, and any tokens, or record of the grouping are discarded.

2.2.2 Regex Lifetime Limits

Lifetime limits are extremely simple in design, a regex process is allowed only a set amount of time in which it can perform its task. Failure to meet this leading to the process being killed.

2.2.3 Sanitisation of input

This solution to Regex patterns does go very much against the reasoning of using Regex in the first place, but has some valid uses cases:
Consider a user sign-up form on a webpage, the user isn't searching, but using anything but Regex in this setting would be pure nightmare fuel for any developer. A quick search of what patterns to use would yield pages such as regular-expressions.info[5] and you'd quickly realise the rabbit-hole for a suitable Regex statement could be:

```
\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}\b
```

But just as easily, it could be:

```
\A(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*
 |  "(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]
      |  \\[\x01-\x09\x0b\x0c\x0e-\x7f])*")
@ (?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?
  |  \[(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}
       (?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?|[a-z0-9-]*[a-z0-9]:
          (?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f]
          |  \\[\x01-\x09\x0b\x0c\x0e-\x7f])+)
     \])\z
```

As suggested by Goyvaerts[5]:

> So even when following official standards, there are still trade-offs to be made. Don't blindly copy regular expressions from online libraries or discussion forums. Always test them on your own data and with your own applications.

As shown by a simple python script:

```python
import re
import time

if __name__ == '__main__':

    email = 'c3146220@uon.edu.au'
    simple_pattern = r'\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}\b'
    complex_pattern = r'^(?=[A-Z0-9._%+-]{6,254}$)[A-Z0-9._%+-]{1,64}@(?:[A
        ↪ -Z0-9-]{1,63}\.){1,8}[A-Z]{2,63}$'

    start_time = time.time()
    re.match(simple_pattern, email, flags=0)
    computation_time = time.time() - start_time
    print(computation_time)

    start_time = time.time()
    re.match(complex_pattern, email, flags=0)
    computation_time = time.time() - start_time
    print(computation_time)
```

A simple checking of group bounds alone costs almost twice the time to be executed:

- Simple match: 0.0001652240753173828 seconds
- Complex match: 0.00030231475830078125 seconds

3 Modern Language Issues

4 Mature Language Issues

References

[1] R. C. Seacord, *Secure Coding in C and C++*. Pearson Education, 2005. [Online]. Available: https://www.pearson.com/us/higher-education/program/Seacord-Secure-Coding-in-C-and-C-2nd-Edition/PGM142190.html.

[2] J. Steinberg. (2014). Sony breach, [Online]. Available: https://www.forbes.com/sites/josephsteinberg/2014/12/11/massive-security-breach-at-sony-heres-what-you-need-to-know/.

[3] D. A. Wheeler, *Secure Programming HOWTO*. David A. Wheeler, 2015. [Online]. Available: https://dwheeler.com/secure-programs/Secure-Programs-HOWTO.pdf.

[4] FTC. (2017). Equifax breach, [Online]. Available: https://www.ftc.gov/equifax-data-breach.

[5] J. Goyvaerts. (2017). Regular expression denial of service - redos, [Online]. Available: https://www.regular-expressions.info/email.html.

[6] Various. (2017). Regular expression denial of service - redos, [Online]. Available: https://www.owasp.org/index.php/Regular_expression_Denial_of_Service_-_ReDoS.

[7] T. Hunt. (2018). Https is easy, [Online]. Available: https://www.troyhunt.com/https-is-easy/.

[8] Various. (2018). Electron, [Online]. Available: https://electronjs.org/.

[9] ——, (2018). Lets encrypt, [Online]. Available: https://letsencrypt.org/.