Secure Coding Practices:
A General Outline

Jay Rovacsek c3146220@uon.edu.au

October 31, 2018

CONTENTS

I. Preface

A. Running with Scissors

Admittedly, the title for this section is very much thanks to one of the first items[5] 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[21]. That is, not to suggest that the concerns of application security has not been discussed for a number of decades[1]

Security as a serious concern is only just now becoming much more "mainstream" to companies than it had previously been[2], movements pushing HTTPS such as Lets Encrypt[22] or high profile individuals such as Troy Hunt[19] 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[4] such as recent breaches in: Sony[7], Equifax[12] and a number of other recent high profile breaches of modern history.

B. 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:[8] (Page 8)

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

II. What Can Go Wrong?

Most developers aren't developers to focus on security, they are creating solutions to problems in order to allow for some benefit to the owner of the software, this could be entertainment, productivity, sales increases or many other reasons.

But security as a base concept is more important than ever ever due to the nature of our increasingly connected world[23]. Interestingly enough the attitude I've encountered within a number of development circles could be described as poor at best;

> "Security is a block to my creative outlet, security just gets in the way"

I won't cite the source of this statement, but I did hear this from a paid software developer before and it has stuck with me since.

A. Recent Prolific Breaches Caused By Poor Development Habits

A number of recent breaches can be used as good examples of why this topic is highly important:

- Equifax Breach 2017 (Estimated 143M individuals records leaked)[14] Caused by CVE-2017-5638[17] Python POC[11]
- Yahoo Breach 2014 (Over 500M Accounts Compromised)[10] Caused by a mixture of unhashed/salted secrets in DB and using insecure hashing algorithm (MD5) easily broken by multiple toolsets[24]
- Uber Breach 2016 (57M Users and 600,000 Drivers Data Exposed)[15]
- Nintendo Switch Jailbreak, uses CVE-2016-4657[9] to execute arbitrary code[16].

III. Common Issues in most Languages

A. Injection

No wide-spread languages in use currently offer an 'out of the box' level of protection against maliciously crafted input by users, generally these issues are mitigated or defended against decently by developers; input usually will have some level of constraint on the data. This however isn't to say that there are better methods to handling user input.

Methods of defense will generally point to either parametrization or limitation of scope with the input data.

1) Attack Vectors: SQL will generally be the best example of this issue, legacy systems will not have the ability to use APIs offered by the language the program was written in or have attempts custom written by the original author that could have missed some potential vectors.

Consider the following:

```
String query = "SELECT * FROM Login WHERE loginId='" +
    ↪ request.getParameter("id") + "'";
```

With this given code, an attacker can easily suggest their username is: ' or '1'='1. which in this case should return all user accounts irrespective of if the user has access or not to this data.

2) Prevention: Prevention as suggested above is trivial, some, many or best of all, all of the following suggestions would be recommended:

- Validation of Input (Can be fallible in a number of cases)
- Parametrization of all non-trusted zones, even better of all zones.
- Use of Stored Procedures
- Limitation of return data in queries.

B. ReDoS / Regex Denial Of Service

Given the requirement for string filtering is extremely common in all applications, ReDoS or Regex Denial Of Service[18] 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)

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) 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.

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[13] 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[13]:

> "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 the checking of group bounds alone costs almost twice the time to be executed and realistically does not give the application any more weight in being more accurate.

	Time Taken (Seconds)
Simple Match	0.0001652240753173828
Complex Match	0.00030231475830078125

While these values don't seem extremely costly, they can quickly become extremely heavy in terms of computation required.

IV. Modern Language Issues

A. Issues Common in Python

Python recently proved to be one of the fastest growing languages based on responses to the stackoverflow developer survey[20] and while python is close to twenty years old, it is considered a new language compared to the likes of C or Perl.

As suggested by Wheeler[8] a number of issues are inherent using python:

1) Lack Of Compile Time Checking: Issues related to the lack of compile time checking in python are a serious security concern as standard static analysis tools for auditing code for security issues have a much more difficult time determining the suitability of code.
It must be noted however, that unit tests and dynamic analysis / fuzzing can prove to be an excellent method to check python code for issues.

B. Issues Common in PHP

PHP is not my area of expertise, however a large number of recent exploits using flaws in PHP have become a trend with the uptake of content generators such as WordPress.

1) Injection: register_globals = ON: Register globals is a common resolution to many issues in developing PHP, however commonly it is left enabled in development or due to inexperience assumed to be okay to leave set on,

The issue lies in any quest that depends on one of these globals set, say we are checking if a user is an admin or not before allowing access to a section/page:

```
if($admin)
{
// let them in
}
else
{
// kick them out
}
```

[3]

Issues arise when a user can inject this global into the request via:

```
script.php?admin=1
```

As suggested again by Wheeler[8], using version 4.2.0 of PHP or greater has features enabled that mitigate most of the danger around this issue.

V. Mature Language Issues

A. C/C++

1) Overflow/Underflow Issues: C and C++ are notorious for the issue of overflows, overflows can easily cause potential vectors for malicious entities to perform a number of attacks. The most common attacks on overflow issues include arbitrary code execution, a basic example thanks to lapk[6]:

```
#include <iostream>
int main( void )
{
int authentication = 0;
char cUsername[ 10 ];
char cPassword[ 10 ];

std::cout << "Username: ";
std::cin >> cUsername;

std::cout << "Pass: ";
std::cin >> cPassword;

if( std::strcmp( cUsername, "admin" ) == 0 &&
    ↪ std::strcmp( cPassword, "adminpass" ) ==
    ↪ 0 )
{
authentication = 1;
}
if( authentication )
{
std::cout << "Access granted\n";
std::cout << ( char )authentication;
}
else
{
std::cout << "Wrong username and password\n";
}

return ( 0 );
}
```

Where in this example, no checking of input bounds is performed when the user has entered data that exceeds the size allocated to username (Char[10]) so as suggested in lapk's[6] example, the input of "0123456789abcdef1" would easily overflow the bounds of the referenced memory. In the case of a compiled x64 binary compiled for Windows the above code would set authentication to 1 despite the comparisons made between password and the expected value.
What is curious about the above example, is it also exhibits two other blatant development flaws:

- Secrets within source code - easily avoided when stored in a DB or loaded from a file outside of source control
- Failure to set authentication = 0 / false when the failure of comparison occurred.

While it must be noted that the example is more for educational purposes than how an application should be written.

References

[1] H. A. S. Booysena and J. H. P. Eloff, "A methodology for the development of secure application systems", in *Information Security — the Next Decade: Proceedings of the IFIP TC11 eleventh international conference on information security, IFIP/Sec '95*, J. H. P. Eloff and S. H. von Solms, Eds. Boston, MA: Springer US, 1995, pp. 255–269, ISBN: 978-0-387-34873-5. DOI: 10.1007/978-0-387-34873-5_20. [Online]. Available: https://doi.org/10.1007/978-0-387-34873-5_20.

[2] M. Howard, "Building more secure software with improved development processes", *IEEE Security & Privacy*, vol. 2, no. 6, pp. 63–65, 2004.

[3] seo-admin. (2004). Php security mistakes, [Online]. Available: http://www.devshed.com/c/a/php/php-security-mistakes/.

[4] A. Apvrille and M. Pourzandi, "Secure software development by example", *IEEE Security & Privacy*, vol. 3, no. 4, pp. 10–17, 2005.

[5] 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.

[6] (2012). C++ buffer overflow, [Online]. Available: https://stackoverflow.com/questions/8782852/c-buffer-overflow.

[7] 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/.

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

[9] Various. (2016). Cve-2016-4657 details, [Online]. Available: https://www.cvedetails.com/cve/CVE-2016-4657.

[10] Yahoo. (2016). Yahoo security notice december 14, 2016, [Online]. Available: https://help.yahoo.com/kb/sln28092.html.

[11] M. Ahmed. (2017). Struts-pwn, [Online]. Available: https://github.com/mazen160/struts-pwn.

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

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

[14] S. Gressin. (2017). The equifax data breach: What to do, [Online]. Available: https://www.consumer.ftc.gov/blog/2017/09/equifax-data-breach-what-do.

[15] D. Khosrowshahi. (2017). 2016 data security incident, [Online]. Available: https://www.uber.com/newsroom/2016-data-incident/.

[16] LiveOverflow. (2017). Lo_nintendoswitch, [Online]. Available: https://github.com/LiveOverflow/lo_nintendoswitch.

[17] Various. (2017). Cve-2017-5638 details, [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2017-5638.

[18] ——, (2017). Regular expression denial of service - redos, [Online]. Available: https://www.owasp.org/index.php/Regular_expression_Denial_of_Service_-_ReDoS.

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

[20] (2018). Stack overflow developer survey results 2018, [Online]. Available: https://insights.stackoverflow.com/survey/2018/#technology.

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

[22] ——, (2018). Lets encrypt, [Online]. Available: https://letsencrypt.org/.

[23] M. Misovski, C. Soboh, and A. Panagiotis, "Secure software development",

[24] Various. (). John the ripper, [Online]. Available: https://github.com/magnumripper/JohnTheRipper.

APPENDIX

A. Proof Of Concept Code

1) Python ReDoS: _

```
#!/usr/bin/env python3.7

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)
```