COMP 4384 Software Security

Module 3: *The Software Security Problem*

Ahmed Tamrawi

 atamrawi  atamrawi.github.io  ahmedtamrawi@gmail.com

```java
public class Puzzle2 {

        public static void main(String[] args) {
                System.out.print("iexplore:");
                http://www.google.com;
                System.out.println(":maximize");
        }

}
```

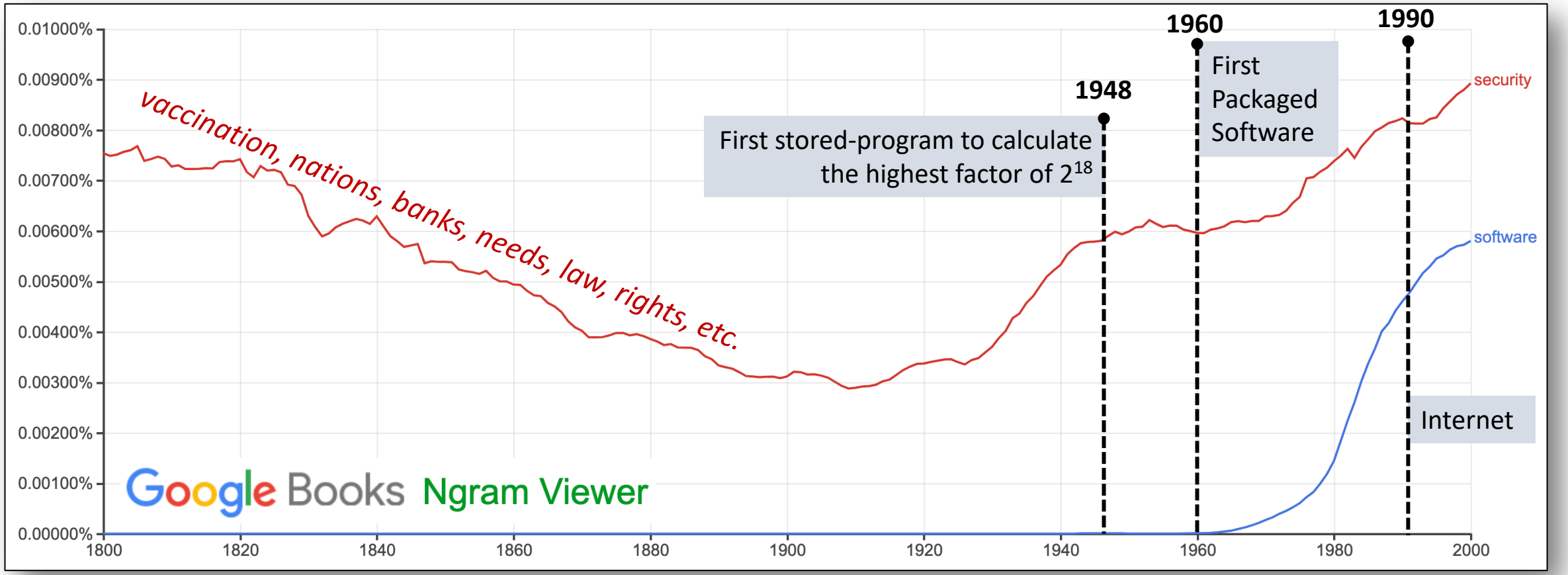
https://gist.github.com/atamrawi/aa1147ef8161349f8276a05f849559c4

What is Software Security?

the state of being free from danger or threat

What is Software Security?

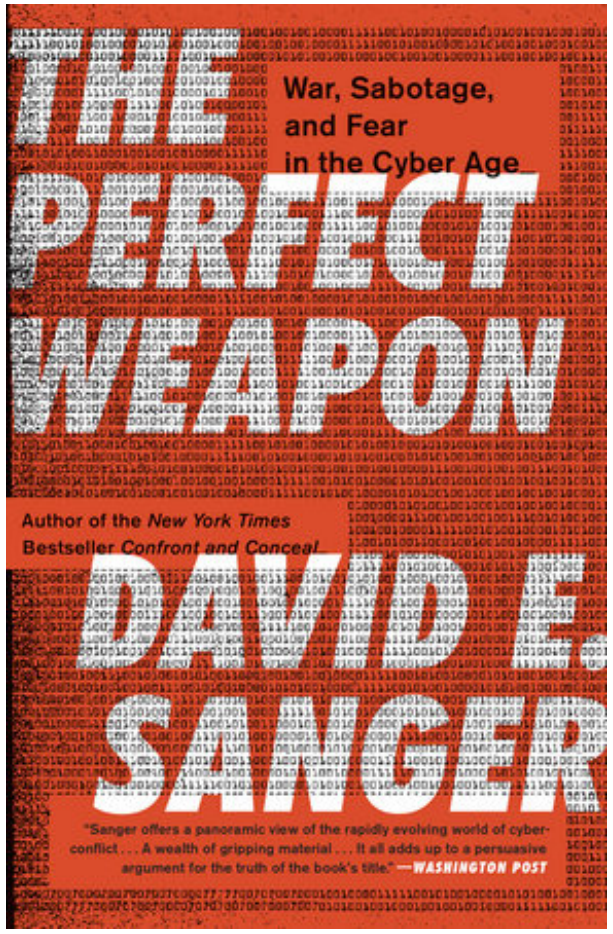
set of instructions, data or programs used to operate computers and execute specific tasks


0.01000%
0.00900%
0.00800%
0.00700%
0.00600%
0.00500%
0.00400%
0.00300%
0.00200%
0.00100%
0.00000%
1800
1820
1840
1860
1880
1900
1920
1940
1960
1980
2000
vaccination, nations, banks, needs, law, rights, etc.
1948
First stored-program to calculate the highest factor of 2^{18}
1960
First Packaged Software
1990
Internet
security
software
Google Books Ngram Viewer

What is Software Security?

is the umbrella term used to describe software that is engineered such that it ***continues to function correctly under malicious attack***

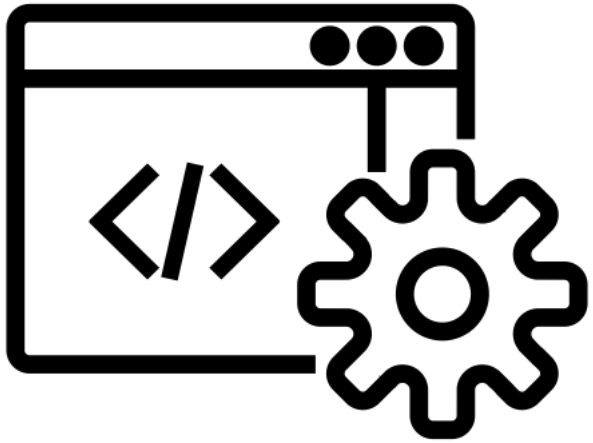
Why do we need Software Security?



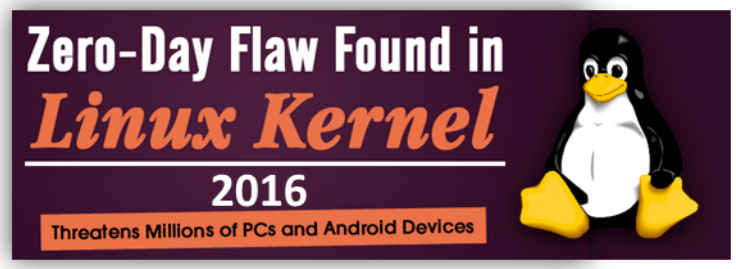
"Each technology goes through a cycle of development and weaponization, ***followed only later*** *by the formulation of doctrine and occasionally by efforts to control the weapon's use."*

WEAPONIZED INTERNET

The **Internet technology** has developed rapidly and it is now being *weaponized* to **sabotage** the electronic or physical assets of an adversary!



Software is an integral part of nearly all technology and almost all prominent attacks on cyber physical systems (CPS) have **exploited vulnerabilities** *rooted in the underlying systems software*.



Zero-Day Flaw Linux
Taking control and privacy

NASA - Mariner 1
$18 million

Android Lollipop
https://threatpost.com/google-aware-of-memory-leakage-issue-in-android-5-1-fix-forthcoming/111640/

Car Recalls - $3 Billion

Knight Capital Trading
$440 million

July 21, 2015

Jeep remotely hijacked

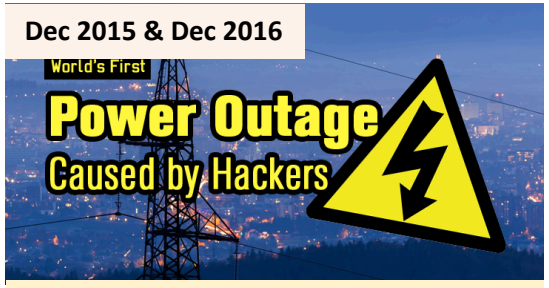
November 29, 2011

HP printers remotely set on fire

August 17, 2009

Destruction Sayano-Shushenskaya Hydroelectric Power Plant

Dec 2015 & Dec 2016



Ukraine power grid attacks

Deployed in 2005, Identified in 2010

STUXnet Worm

August 2003

Northeast Power Blackout

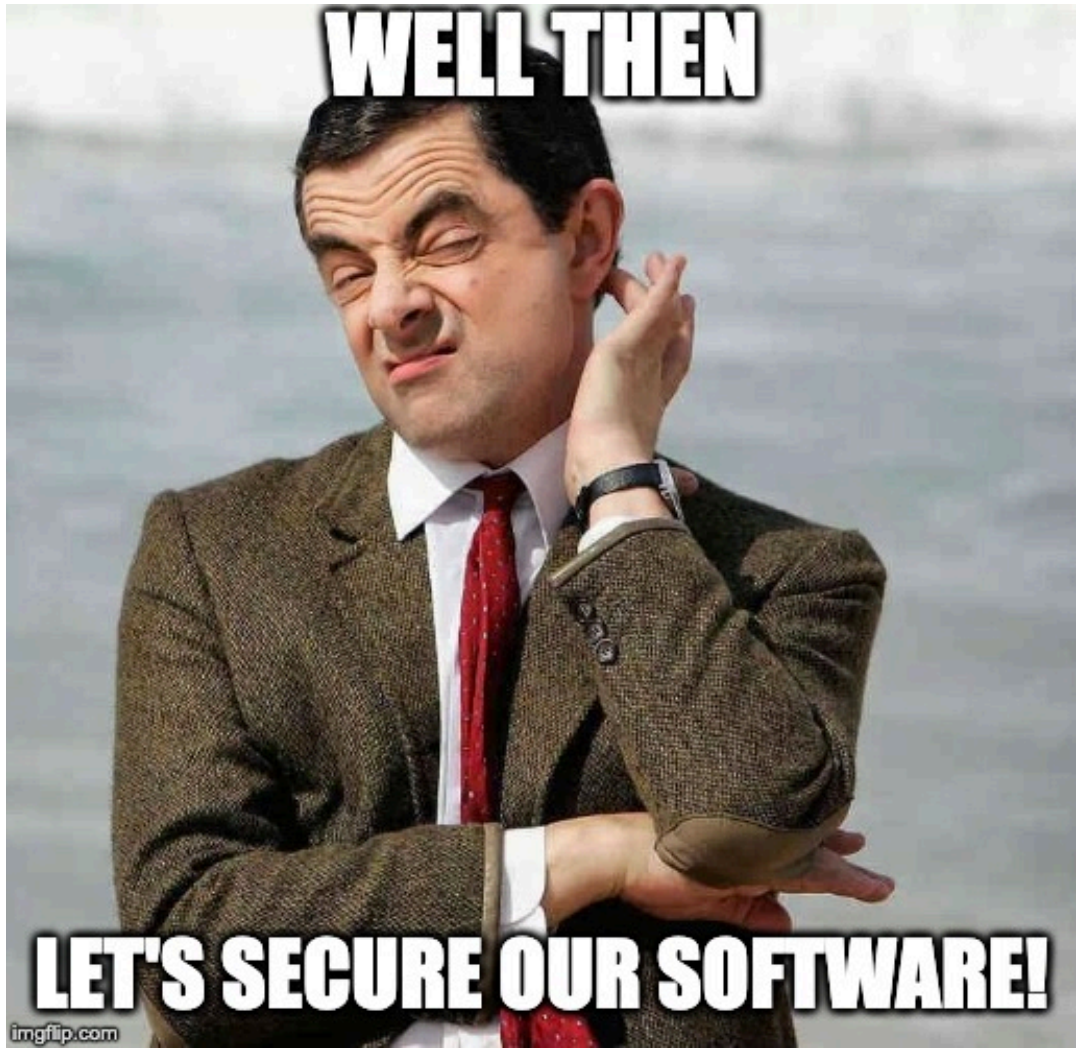
August 2003

Davis-Besse Nuclear Power Plant

No need for bombs, ***Plant Malware!***

is investing billions of dollars into Securing Software

APAC	*Automated Program Analysis for Cybersecurity*
VET	*Vetting Commodity IT Software and Firmware*
HACMS	*High Assurance Cyber Military Systems*
STAC	*Space/Time Analysis for Cybersecurity*
CASE	*Cyber Assured Systems Engineering*
CHESS	*Computers and Humans Exploring Software Security*
ARCOS	*Automated Rapid Certification Of Software*


WELL THEN
LET'S SECURE OUR SOFTWARE!
imgflip.com

Software Security

- On the first day of class, mechanical engineers learn a critical lesson: *Pay attention and learn this stuff, or the bridge you build could fall down*.
- By contrast, on the first day of software engineering class, budding developers are taught that they can build anything that they can dream of. They usually start with “hello world.”

```c
#include <stdio.h>

int main() {
  printf("Hello World\n");
  return 0;
}
```

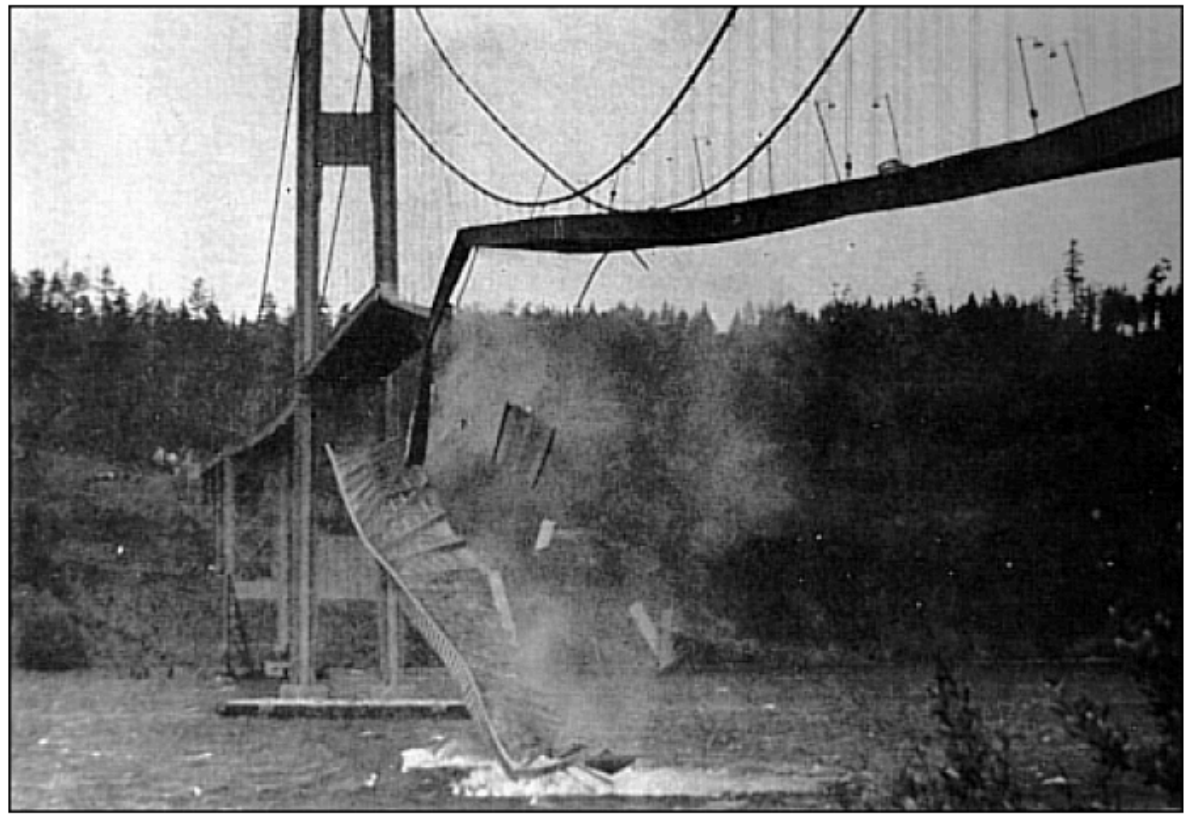


Figure 1 A 600-foot section of the Tacoma Narrows bridge crashes into Puget Sound as the bridge twists and torques itself to death. Mechanical engineers are warned early on that this can happen if they don't practice good engineering.

Software Security

- An overly optimistic approach to software development has certainly led to the **creation of some mind-boggling stuff**, but it has likewise allowed us to paint ourselves into the corner from a **security perspective**.
- Simply put, we neglected to think about what would happen to our **software if it were intentionally and maliciously attacked**.
- Much of today's software is so **fragile** that it **barely functions properly when its environment is pristine and predictable**.
- If the environment in which our **fragile software** runs turns out to be pugnacious and pernicious (as much of the Internet environment turns out to be), software fails spectacularly, splashing into the metaphorical Puget Sound.

Software Security

- The biggest problem in computer security today is that most systems aren't **constructed with security in mind**.
- **Reactive network technologies** such as firewalls can help alleviate obvious **script kiddie attacks** on servers, but they do nothing to address the real security problem: bad software.
- If we want to solve the computer security problem, we need to do more to build secure software.
- **Software security** is the practice of building software to be secure and function properly under malicious attack.

The Real Problem

- The problem is that most developers **have little idea what bugs to look for, or what to do about bugs if they do find them**.
- We live in a time of unprecedented economic growth, increasingly fueled by computer and communications technology.
- We use software to automate factories, streamline commerce, and put information into the hands of people who can act upon it.
- We live in the information age, and software is the primary means by which we tame information.

The Real Problem

- Virus scanners, firewalls, patch management, and intrusion detection systems are all means by which we make up for shortcomings in software security.
- The software industry puts **more effort into compensating for bad security** than it puts into creating secure software in the first place.
- Just as every ship should have lifeboats, it is both good and healthy that our industry creates ways to quickly compensate for a newly discovered vulnerability.
- **Changing the state of software security requires changing the way software is built. This is not an easy task.**

The Software Security Problem

- We believe that the most effective way to improve software security is to study past security errors and prevent them from happening in the future.
- Our philosophy is similar to that of Henry Petroski: *To build a strong system, you have to understand how the system is likely to fail* [Petroski, 1985]
- Mistakes are **inevitable**, but you have a measure of control over your mistakes. Although you can't have precise knowledge of your next blunder, you can control the set of possibilities.

The Software Security Problem

- Being aware of common pitfalls might sound like a good way to avoid falling prey to them, but awareness by itself often proves to be insufficient.
- Children learn the spelling rule “i before e except after c,” but widespread knowledge of the rule does not prevent **believe** from being a commonly misspelled word.
- **Understanding security is one thing; applying your understanding in a complete and consistent fashion to meet your security goals is quite another**.

Defensive Programming Is Not Enough

- Defensive programming refers to *the practice of coding with the mindset that errors are inevitable and that, sooner or later, something will go wrong and lead to unexpected conditions within the program*.
- Kernighan and Plauger call it "*writing the program so it can cope with small disasters*" [Kernighan and Plauger, 1981].
- Good defensive programming requires adding code to check one's assumptions.
- Good defensive programming makes bugs both easier to find and easier to diagnose.

Defensive Programming Is Not Enough

- But defensive programming **does not guarantee secure software** (although the notion of expecting anomalies is very much a step in the right direction).
- When we talk about security, we assume the existence of an adversary— someone who is intentionally trying to subvert the system.
- Instead of trying to compensate for typical kinds of accidents (on the part of either the programmer or the user), software security is about creating programs that *behave correctly even in the presence of malicious behavior*.

Defensive Programming Is Not Enough

Implement function `printMsg` that accepts two arguments from the user:

- `file`: A pointer of type FILE.
- `msg`: A string message.

`printMsg` writes the given `msg` to `file` and flushes the writing buffer.

Defensive Programming Is Not Enough

Implement function `printMsg` that accepts two arguments from the user:

- `file`: A pointer of type FILE.
- `msg`: A string message.

`printMsg` writes the given `msg` to `file` and flushes the writing buffer.

Although, the implemented program does not **violate any of the requirements**, the program will **crash** If either argument is NULL.

```c
#include <stdio.h>

void printMsg(FILE* file, char* msg) {
        fprintf(file, msg);
        fflush(file);
}

int main(int argc, char** argv) {
        FILE *file = fopen(argv[1], "w");
        char* msg = argv[2];
        printMsg(file, msg);
        return 0;
}
```

Defensive Programming Is Not Enough

Implement function `printMsg` that accepts two arguments from the user:

- `file`: A pointer of type FILE.
- `msg`: A string message.

`printMsg` writes the given `msg` to `file` and flushes the writing buffer.

Apply a **defensive programming** approach to avoid program crashes fi either argument is NULL

```c
#include <stdio.h>

void printMsg(FILE* file, char* msg) {
    if(file == NULL) {
        printf("%s", "File is NULL\n");
    } else if(msg == NULL) {
        printf("%s", "Message is NULL\n");
    } else {
        fprintf(file, msg);
        fflush(file);
    }
}

int main(int argc, char** argv) {
    if(argc < 2) {
        return 0;
    }
    FILE *file = fopen(argv[1], "w");
    char* msg = argv[2];
    printMsg(file, msg);
    return 0;
}
```

Defensive Programming Is Not Enough

Implement function `printMsg` that accepts two arguments from the user:

- `file`: A pointer of type FILE.
- `msg`: A string message.

`printMsg` writes the given `msg` to `file` and flushes the writing buffer.

Apply a **defensive programming** approach to avoid program crashes fi either argument is NULL

From a security perspective, these checks simply **do not go far enough**

```c
#include <stdio.h>

void printMsg(FILE* file, char* msg) {
    if(file == NULL) {
        printf("%s", "File is NULL\n");
    } else if(msg == NULL) {
        printf("%s", "Message is NULL\n");
    } else {
        fprintf(file, msg);
        fflush(file);
    }
}

int main(int argc, char** argv) {
    if(argc < 2) {
        return 0;
    }
    FILE *file = fopen(argv[1], "w");
    char* msg = argv[2];
    printMsg(file, msg);
    return 0;
}
```

Defensive Programming Is Not Enough

Implement function `printMsg` that accepts two arguments from the user:

- `file`: A pointer of type FILE.
- `msg`: A string message.

`printMsg` writes the given `msg` to `file` and flushes the writing buffer.

From a security perspective, these checks simply **do not go far enough**

Although we have prevented a caller from **crashing the program** by providing NULL values, the code does not account for the fact that the value of the **`msg`** parameter itself might be malicious.

```c
#include <stdio.h>

void printMsg(FILE* file, char* msg) {
    if(file == NULL) {
        printf("%s", "File is NULL\n");
    } else if(msg == NULL) {
        printf("%s", "Message is NULL\n");
    } else {
        fprintf(file, msg);
        fflush(file);
    }
}

int main(int argc, char** argv) {
    if(argc < 2) {
        return 0;
    }
    FILE *file = fopen(argv[1], "w");
    char* msg = argv[2];
    printMsg(file, msg);
    return 0;
}
```

Similar to `printf`

`fprintf` - Write formatted data to stream

specifier	Output	Example
`d` *or* `i`	Signed decimal integer	392
`u`	Unsigned decimal integer	7235
`o`	Unsigned octal	610
`x`	Unsigned hexadecimal integer	7fa
`X`	Unsigned hexadecimal integer (uppercase)	7FA
`f`	Decimal floating point, lowercase	392.65
`F`	Decimal floating point, uppercase	392.65
`e`	Scientific notation (mantissa/exponent), lowercase	3.9265e+2
`E`	Scientific notation (mantissa/exponent), uppercase	3.9265E+2
`g`	Use the shortest representation: `%e` or `%f`	392.65
`G`	Use the shortest representation: `%E` or `%F`	392.65
`a`	Hexadecimal floating point, lowercase	-0xc.90fep-2
`A`	Hexadecimal floating point, uppercase	-0XC.90FEP-2
`c`	Character	a
`s`	String of characters	sample
`p`	Pointer address	b8000000
`n`	Nothing printed. The corresponding argument must be a pointer to a `signed int`. The number of characters written so far is stored in the pointed location.	
`%`	A `%` followed by another `%` character will write a single `%` to the stream.	%

```c
/* printf example */
#include <stdio.h>

int main()
{
   printf ("Characters: %c %c \n", 'a', 65);
   printf ("Decimals: %d %ld\n", 1977, 650000L);
   printf ("Preceding with blanks: %10d \n", 1977);
   printf ("Preceding with zeros: %010d \n", 1977);
   printf ("Some different radices: %d %x %o %#x %#o \n", 100, 100, 100, 100, 100);
   printf ("floats: %4.2f %+.0e %E \n", 3.1416, 3.1416, 3.1416);
   printf ("Width trick: %*d \n", 5, 10);
   printf ("%s \n", "A string");
   return 0;
}
```

```
Characters: a A
Decimals: 1977 650000
Preceding with blanks:       1977
Preceding with zeros: 0000001977
Some different radices: 100 64 144 0x64 0144
floats: 3.14 +3e+000 3.141600E+000
Width trick:    10
A string
```

fmt_part2.c

```c
#include <stdio.h>

void printMsg(FILE* file, char* msg) {
        if(file == NULL) {
                printf("%s", "File is NULL\n");
        } else if(msg == NULL) {
                printf("%s", "Message is NULL\n");
        } else {
                fprintf(file, msg);
                fflush(file);
        }
}

int main(int argc, char** argv) {
        if(argc < 2) {
                return 0;
        }
        FILE *file = fopen(argv[1], "w");
        char* msg = argv[2];
        printMsg(file, msg);
        return 0;
}
```

Disable Address space layout randomization (ASLR)
https://en.wikipedia.org/wiki/Address_space_layout_randomization

```
ahmed@Ubuntu-Machine: ~/Desktop/software-security
ahmed@Ubuntu-Machine:~/Desktop/software-security$ sudo sysctl kernel.randomize_va_space=0
kernel.randomize_va_space = 0
ahmed@Ubuntu-Machine:~/Desktop/software-security$ gcc fmt_part2.c -o fmt_part2
fmt_part2.c: In function ‘printMsg’:
fmt_part2.c:8:3: warning: format not a string literal and no format arguments [-Wformat-security]
    8 |   fprintf(file, msg);
      |   ^~~~~~~
ahmed@Ubuntu-Machine:~/Desktop/software-security$ ./fmt_part2 hello.txt AAA$(perl -e 'print "%p."x190')
ahmed@Ubuntu-Machine:~/Desktop/software-security$
```

Compile/Build the program

Run the program with malicious content

```
AAA%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p
.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%
p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.
%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p
.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%
p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.
```

home > ahmed > Desktop > software-security > hello.txt

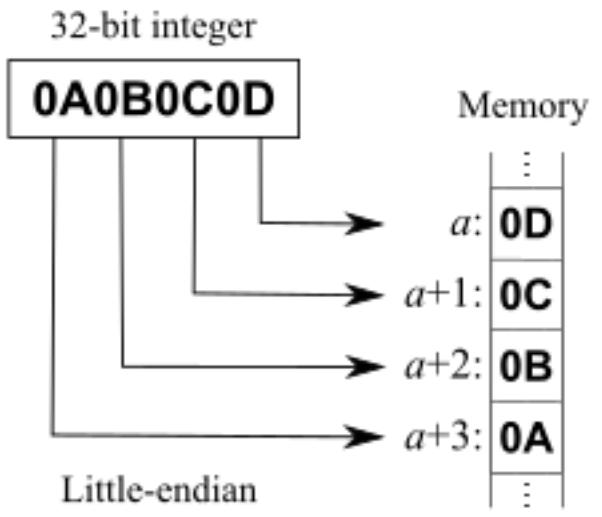
```
AAA0x7ffffffe1ac.0x2.0x4.0x1.0x7ffffffe1ac.0x5555555592a0.0x7fffffffdd70.0x55555555523f.0x7fffffffde68.0x3555550a0.0x5555555592a0.0x7ffffffe1ac.(nil).0x7ffff7ded0b3.
0x7ffff7ffc620.0x7fffffffde68.0x300000000.0x5555555551e2.0x555555555250.0x6bde7266e01f5600.0x5555555550a0.0x7fffffffde60.(nil).(nil).0x94218d995b1f5600.0x94219ddb40d15600.
(nil).(nil).(nil).0x3.0x7fffffffde68.0x7fffffffde88.0x7ffff7ffe190.(nil).(nil).0x5555555550a0.0x7fffffffde60.(nil).(nil).0x5555555550ce.0x7fffffffde58.0x1c.0x3.
0x7ffffffe196.0x7ffffffe1a2.0x7ffffffe1ac.(nil).0x7ffffffe3ea.0x7ffffffe3fa.0x7ffffffe45c.0x7ffffffe46f.0x7ffffffe483.0x7ffffffe4b0.0x7ffffffe4c7.0x7ffffffe4f3.
0x7ffffffe506.0x7ffffffe518.0x7ffffffe52b.0x7ffffffe54b.0x7ffffffe574.0x7ffffffe588.0x7ffffffe59f.0x7ffffffe5b2.0x7ffffffe5ce.0x7ffffffe5f8.0x7ffffffe606.
0x7ffffffe621.0x7ffffffe636.0x7ffffffe66a.0x7ffffffe693.0x7ffffffe6b4.0x7ffffffe6c1.0x7ffffffe6d2.0x7ffffffe6e1.0x7ffffffe6f3.0x7ffffffe6fe.0x7ffffffece0.
0x7fffffffed01.0x7fffffffed12.0x7fffffffed68.0x7fffffffed97.0x7fffffffeda6.0x7fffffffedbd.0x7fffffffedd5.0x7fffffffedf7.0x7fffffffee0e.0x7fffffffee22.0x7fffffffee42.
0x7fffffffee4d.0x7fffffffee6b.0x7fffffffee76.0x7fffffffee7e.0x7fffffffee90.0x7fffffffeeaf.0x7fffffffeec6.0x7fffffffef1b.0x7fffffffef83.0x7fffffffef95.0x7fffffffefcb.
0x7fffffffefde.(nil).0x21.0x7ffff7fce000.0x10.0x178bfbff.0x6.0x1000.0x11.0x64.0x3.0x555555554040.0x4.0x38.0x5.0xd.0x7.0x7ffff7fcf000.0x8.(nil).0x9.0x5555555550a0.0xb.
0x3e8.0xc.0x3e8.0xd.0x3e8.0xe.0x3e8.0x17.(nil).0x19.0x7fffffffe179.0x1a.(nil).0x1f.0x7fffffffe         x7fffffffe189.(nil).(nil).0xc454f0e434a81500.0x35ef3933700fd8.
0x34365f363878ab.0x2f2e000000000000.0x747261705f746d66.0x2e6f6c6c65680032.0x2541414 1         70252e70.0x70252e70252e7025.0x252e70252e70252e.0x2e70252e70252e70.
0x70252e70252e7025.0x252e70252e70252e.0x2e70252e70252e70.0x70252e70252e7025.0x252e70252e7025e         2e70252e70.0x70252e70252e7025.0x252e70252e70252e.0x2e70252e70252e70.
0x70252e70252e7025.0x252e70252e70252e.0x2e70252e70252e70.0x70252e70252e7025.0x252e70252e70252e.0x2e70252e70252e70.0x70252e70252e7025.0x252e70252e70252e.0x2e70252e70252e70.
0x70252e70252e7025.0x252e70252e70252e.0x2e70252e70252e70.0x70252e70252e7025.0x252e70252e70252e.0x2e70252e70252e70.0x70252e70252e7025.0x252e70252e70252e.0x2e70252e70252e70.
0x70252e70252e7025.0x252e70252e70252e.0x2e70252e70252e70.0x70252e70252e7025.0x252e70252e70252e.0x2e70252e70252e70.0x70252e70252e7025.0x252e70252e70252e.0x2e70252e70252e70.
0x70252e70252e7025.0x252e70252e70252e.
```

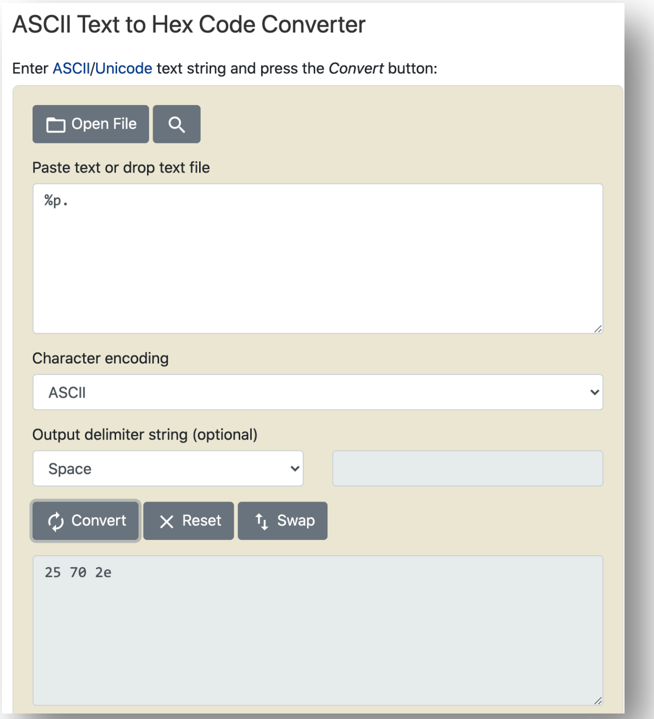
AAA

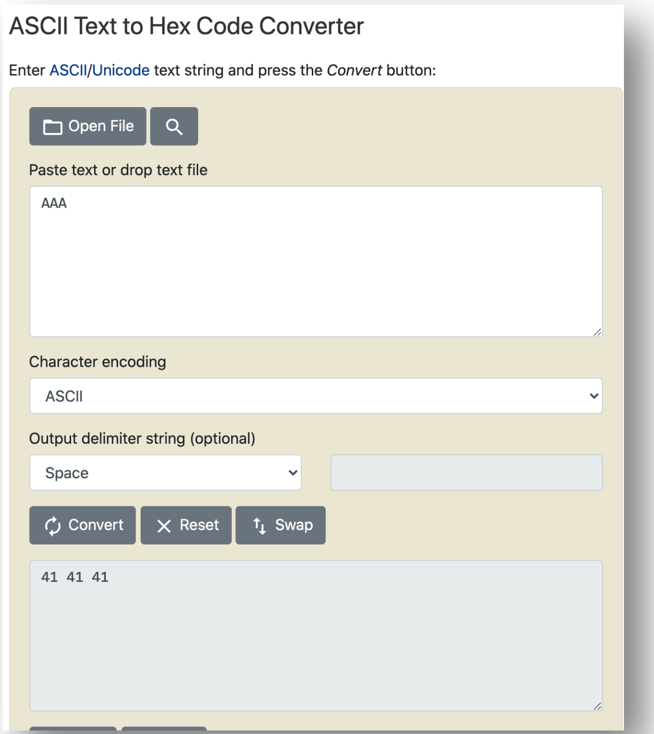
%p.

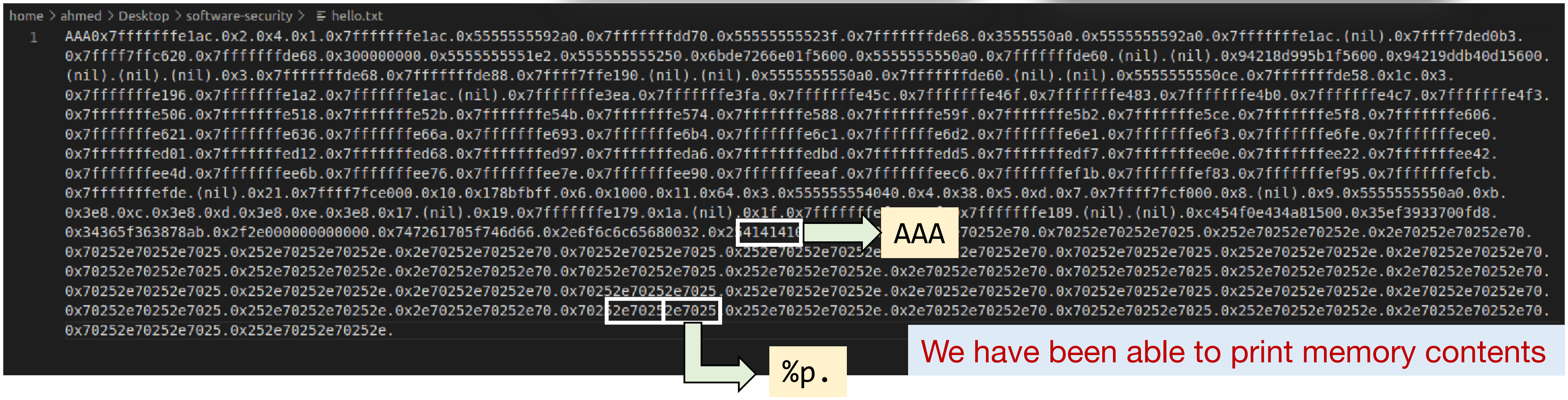
We have been able to print memory contents

Print memory content









We have been able to print memory contents

Defensive Programming Is Not Enough

Implement function `printMsg` that accepts two arguments from the user:

- `file`: A pointer of type FILE.
- `msg`: A string message.

`printMsg` writes the given `msg` to `file` and flushes the writing buffer.

By providing `msg` as the **format string argument** to `fprintf`, the code leaves open the possibility that an attacker could specify a *malicious format string designed to carry out a format string attack*.

```c
#include <stdio.h>

void printMsg(FILE* file, char* msg) {
        if(file == NULL) {
                printf("%s", "File is NULL\n");
        } else if(msg == NULL) {
                printf("%s", "Message is NULL\n");
        } else {
                fprintf(file, msg);
                fflush(file);
        }
}

int main(int argc, char** argv) {
        if(argc < 2) {
                return 0;
        }
        FILE *file = fopen(argv[1], "w");
        char* msg = argv[2];
        printMsg(file, msg);
        return 0;
}
```

Defensive Programming Is Not Enough

- This attempt at defensive programming shows how a straightforward approach to solving a programming problem can turn out to be **insecure**.
- The people who created the programming languages, libraries, frameworks, protocols, and conventions that most programmers build upon did not anticipate all the ways their creations would be assailed.
- Because of a design oversight, format strings became an attack vector, and seemingly reasonable attempts at **error handling turn out to be inadequate in the face of attack**.

Defensive Programming Is Not Enough

- A security-conscious programmer will deprive an attacker of the opportunity this vulnerability represents by supplying a fixed format string.

```c
#include <stdio.h>

void printMsg(FILE* file, char* msg) {
        if(file == NULL) {
                printf("%s", "File is NULL\n");
        } else if(msg == NULL) {
                printf("%s", "Message is NULL\n");
        } else {
                fprintf(file, msg);
                fflush(file);
        }
}

int main(int argc, char** argv) {
        if(argc < 2) {
                return 0;
        }
        FILE *file = fopen(argv[1], "w");
        char* msg = argv[2];
        printMsg(file, msg);
        return 0;
}
```

```c
#include <stdio.h>

void printMsg(FILE* file, char* msg) {
        if(file == NULL) {
                printf("%s", "File is NULL\n");
        } else if(msg == NULL) {
                printf("%s", "Message is NULL\n");
        } else {
                fprintf(file, "%.128s", msg);
                fflush(file);
        }
}

int main(int argc, char** argv) {
        if(argc < 2) {
                return 0;
        }
        FILE *file = fopen(argv[1], "w");
        char* msg = argv[2];
        printMsg(file, msg);
        return 0;
}
```

Defensive Programming Is Not Enough

- In considering the range of things that might go wrong with a piece of code, programmers tend to stick with their experience:
  - The program might crash, it might loop forever, or it might simply fail to produce the desired result.
- All these failure modes are important but preventing them does not lead to software that stands up to attack.
- *This results in code that might be well defended against the types of problems that a programmer is familiar with but that is still easy for an attacker to subvert*.

Security Features != Secure Features

- As Michael Howard, a program manager on the Microsoft Security Engineering Team, says, "**Security features != Secure features**" [Howard and LeBlanc, 2002].
- For a program to be secure, all portions of the program must be secure, not just the bits that explicitly address security.
  - In many cases, security failings are not related to security features at all.
- A security feature can fail and jeopardize system security in plenty of ways, but there are usually many more ways in which defective non-security features can go wrong and lead to a security problem.

Security Features != Secure Features

- Security features are (usually) implemented with the idea that they must function correctly to maintain system security, but **non-security features often fail to receive this same consideration**, even though they are often just as critical to the system's security.
- Imagine a burglar who wants to break into your house. He might start by walking up to the front door and trying to turn the doorknob. If the door is locked, he has run into a security feature. Now imagine that the door's hinges are on the outside of the house.
- The builder probably didn't think about the hinge in relation to security; the hinges are by no means a security feature—they are present so that the door will meet the "easy to open and close" requirement.
- But now it's unlikely that our burglar will spend time trying to pick the lock or pry open the door. He'll simply lift out the hinge bolts and remove the door.

Date	Program	Effect	Reference
March 2002	zLib	Denial of service affecting many programs, including those that display or manipulate PNG files.	http://www.securityfocus.com/bid/6431
November 2002	Internet Explorer	Malicious PNG file can be used to execute arbitrary code when displayed in Internet Explorer.	http://www.microsoft.com/technet/security/bulletin/MS02-066.mspx
August 2004	libPNG	Denial of service affecting users of Firefox, Opera, Safari, and many other programs.	http://www.securityfocus.com/bid/6431
September 2004	MS GDI+	JPG-rendering code that enables the remote execution of arbitrary code. Affects Internet Explorer, Microsoft Office, and other Microsoft products.	http://www.microsoft.com/technet/security/bulletin/MS04-028.mspx
July 2005	zLib	Creates the potential for remote code execution. Affects many programs, including those that display or manipulate PNG files.	http://www.securityfocus.com/bid/14162
December 2005	Windows Graphics Rendering Engine	Rendering of WMF files enables remote code execution of arbitrary code. Exploitable through Internet Explorer.	http://www.microsoft.com/technet/security/bulletin/ms06-001.mspx
January 2007	Java 2 Platform	Rendering of GIF image allows the remote execution of arbitrary code through a hostile applet.	http://www.sunsolve.sun.com/search/document.do?assetkey=1-26-102760-1

Vulnerabilities in image display code over the last five years. All are significant vulnerabilities. None have anything to do with security features.

The Quality Fallacy

- Anyone who has ever written a program knows that mistakes are inevitable.
- Anyone who writes software professionally knows that producing good software requires a **systematic approach to finding bugs**.
- By far the most widely used approach to bug finding is **dynamic testing**, which *involves running the software and comparing its output against an expected result*.

The Quality Fallacy

- Advocates of **extreme programming** want to see a lot of small tests (unit tests) written by the programmer even before the code is written.
- Large software organizations have big groups of dedicated QA engineers who are responsible for nothing other than writing tests, running tests, and evaluating test results.
- It is almost impossible to improve software security merely by improving quality assurance.
- In practice, most software quality efforts are geared toward **testing program functionality**. The purpose is to find the bugs that will affect the most users in the worst ways.

The Quality Fallacy

- **Functionality testing** works well for making sure that typical users with typical needs will be happy, but it just won't work for finding security defects that aren't related to security features.
- Most software testing is aimed at **comparing the implementation to the requirements**, and *this approach is inadequate for finding security problems*.

The Quality Fallacy

- The software (the implementation) has a list of things it's supposed to do (the requirements).
- Imagine testing a piece of software by running down the list of requirements and making sure the implementation fulfills each one.
- If the software fails to meet a particular requirement, you've found a bug. This works well for testing software functionality, even security functionality, but it will miss many security problems.
- Security problems are often **not violations of the requirements**. Instead, security problems are frequently "unintended functionality" that causes the program to be insecure.

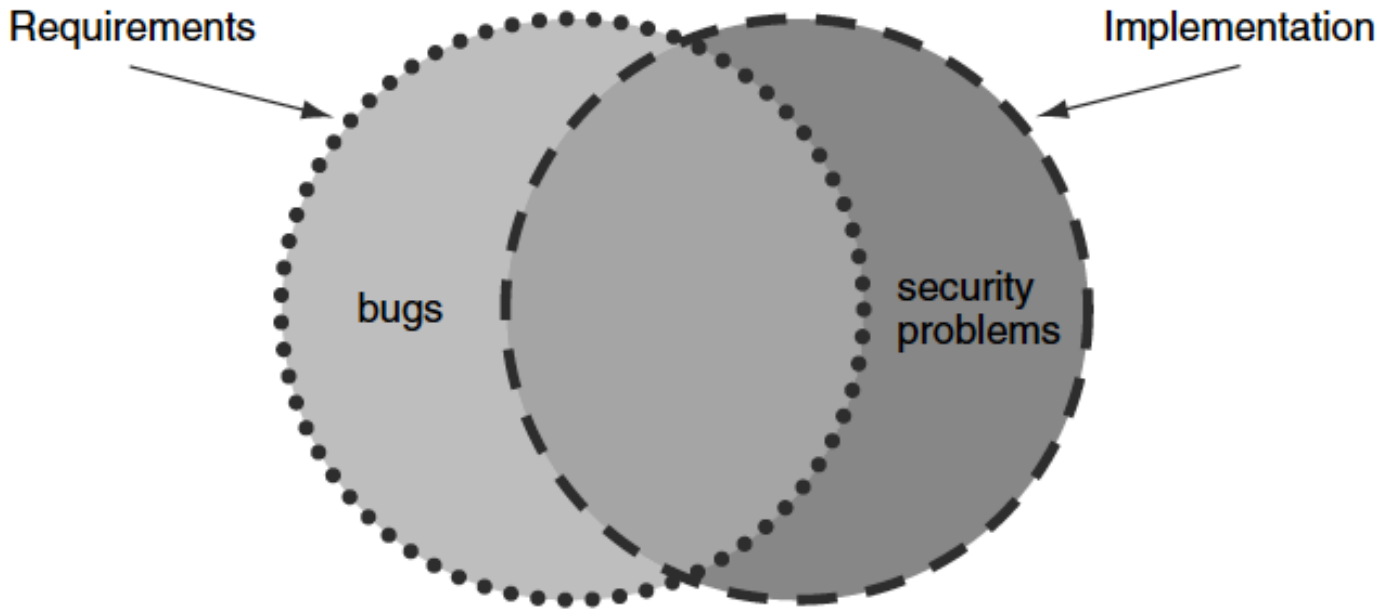
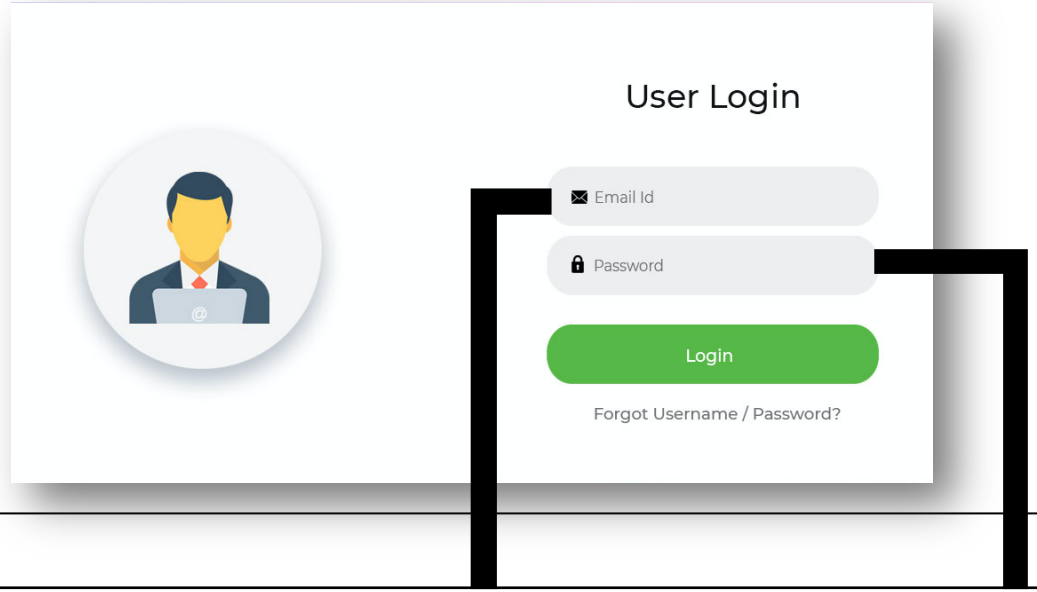


Figure 1.1 Testing to make sure that the implementation includes the features described in the specification will miss many security problems.

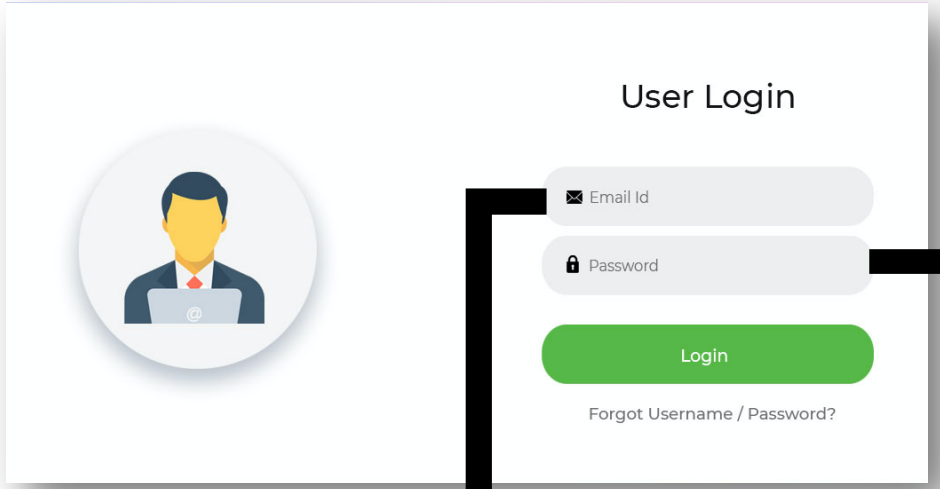
The Quality Fallacy

- Ivan Arce, CTO of Core Security Technologies, put it like this: *Reliable software does what it is supposed to do. Secure software does what it is supposed to do, and nothing else*.



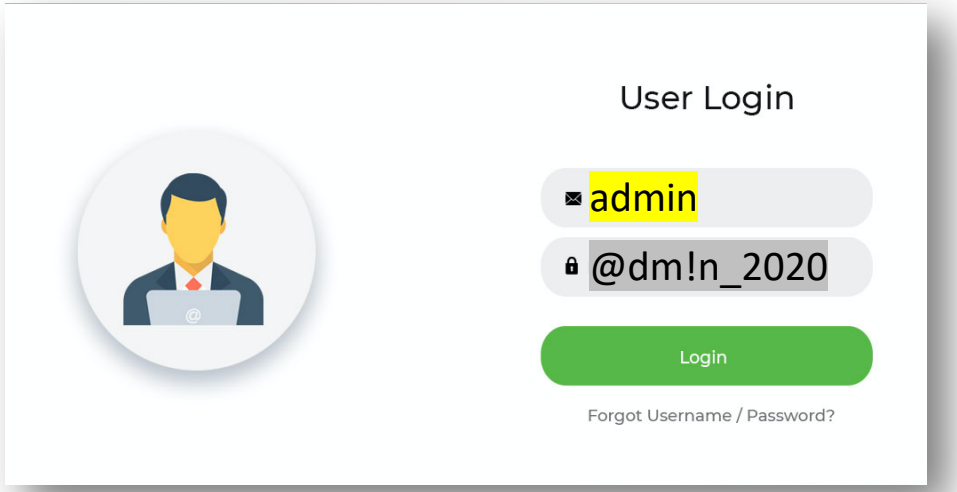

```
$conn = pg_pconnect("dbname=user_accounts");
$result = pg_query(conn, "SELECT * from user_accounts WHERE username='".$_GET['user']."' AND password ='".$_GET['pwd']."'");
if(pg_query_num($result) > 0) {
    echo "Admin logged in";
    admin_control_panel_redirect();
}
```

SQL Command to be executed




```
$conn = pg_pconnect("dbname=user_accounts");
$result = pg_query(conn, "SELECT * from user_accounts WHERE username='".$_GET['user']."' AND password ='".$_GET['pwd']."'");
if(pg_query_num($result) > 0) {
    echo "Admin logged in";
    admin_control_panel_redirect();
}
```

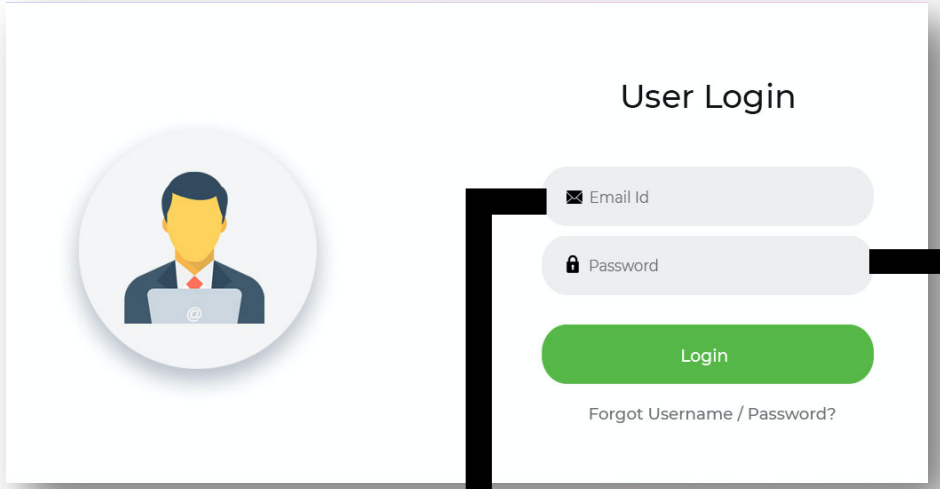
SQL Command to be executed



SELECT * from user_accounts WHERE username='admin' AND password ='@dm!n_2020'

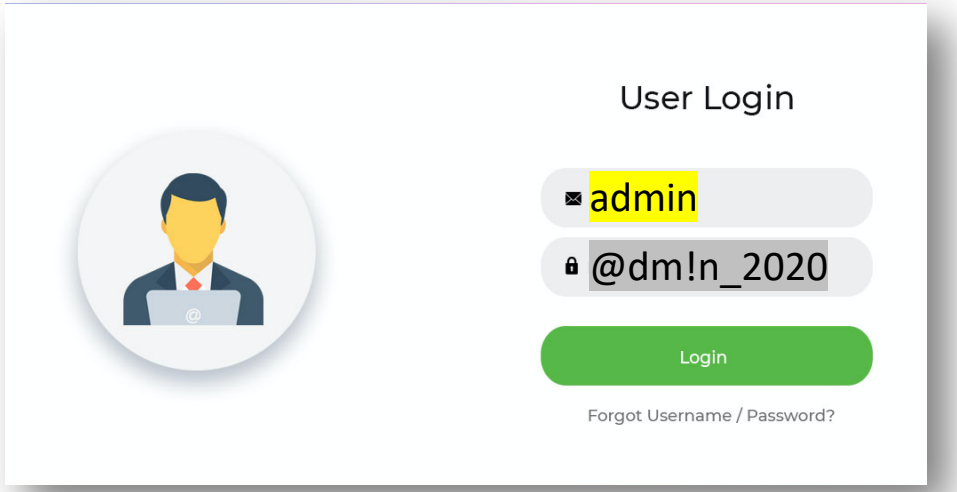
Result: *Redirects the user to the admin control panel page if there is a single match for the passed credentials in user_accounts database*

This code might meet the program's requirements, but it enables an **SQL Injection attack** as it does not sanitize the query parameters provided by the user and so the attacker is free to provide malicious input




```
$conn = pg_pconnect("dbname=user_accounts");
$result = pg_query(conn, "SELECT * from user_accounts WHERE username='".$_GET['user']."' AND password ='".$_GET['pwd']."'");
if(pg_query_num($result) > 0) {
    echo "Admin logged in";
    admin_control_panel_redirect();
}
```

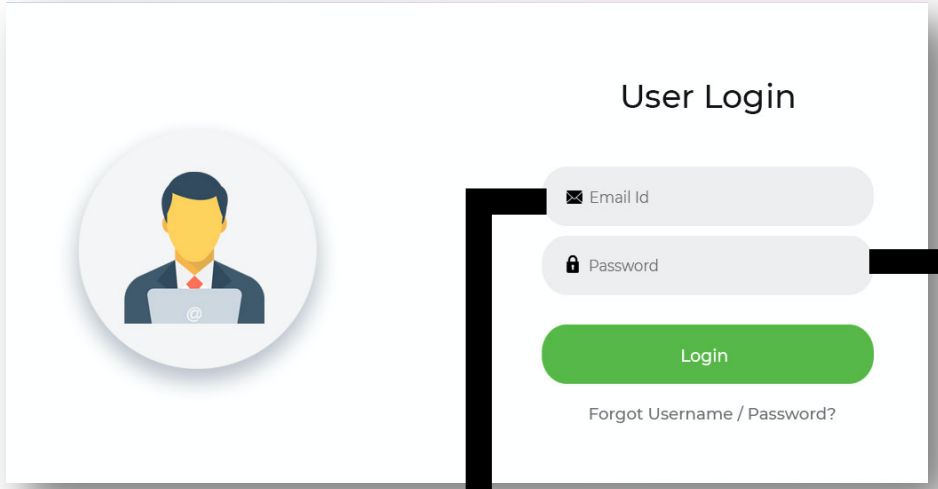
SQL Command to be executed



SELECT * from user_accounts WHERE username='admin' AND password ='@dm!n_2020'

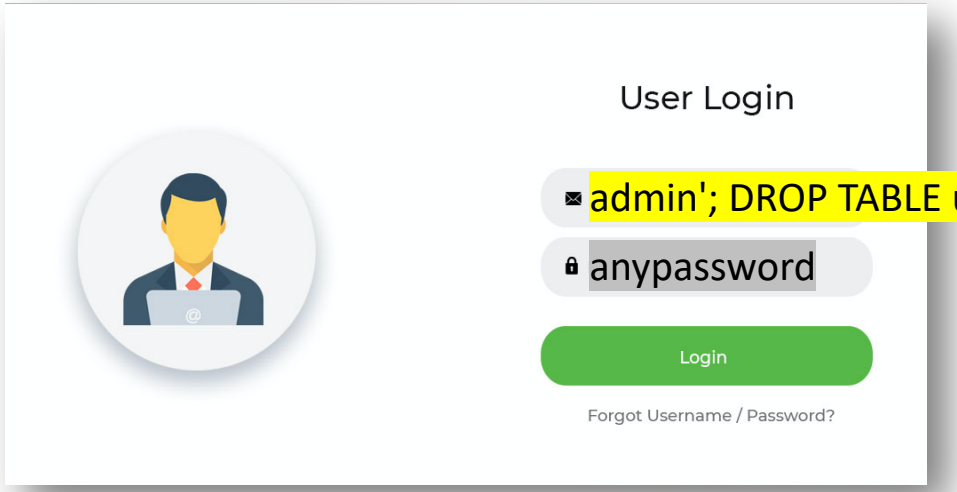
Result: *Redirects the user to the admin control panel page if there is a single match for the passed credentials in user_accounts database*

This code might meet the program's requirements, but it enables an **SQL Injection attack** as it does not sanitize the query parameters provided by the user and so the attacker is free to provide malicious input




```
$conn = pg_pconnect("dbname=user_accounts");
$result = pg_query(conn, "SELECT * from user_accounts WHERE username='".$_GET['user']."' AND password ='".$_GET['pwd']."'");
if(pg_query_num($result) > 0) {
    echo "Admin logged in";
    admin_control_panel_redirect();
}
```

SQL Command to be executed



Comment the rest of the SQL command

SELECT * from user_accounts WHERE username='admin'; DROP TABLE user_accounts #'
AND password ='anypassword'

Result: *Queries all user accounts with username "admin" and drops user_accounts table causing system corruption.*

The Quality Fallacy

- A growing number of organizations attempt to overcome the lack of focus on security by **mandating a penetration test**.
  - After a system is built, testers stage a mock attack on the system.
- A black-box test gives the attackers no information about how the system is constructed. This might sound like a realistic scenario, but in reality, it is **both inadequate and inefficient**.
- Testing cannot begin until the system is complete, and testers have exclusive access to the software only until the release date.
- After the release, attackers and defenders are on equal footing; attackers are now able to test and study the software, too.

The Quality Fallacy

- The narrow window means that the sum total of all attackers can easily have more hours to spend hunting for problems than the defenders have hours for testing.
- The testers eventually move on to other tasks, but attackers get to keep on trying.
- The end result of their greater investment is that attackers can find a greater number of vulnerabilities.

The Quality Fallacy

- Black-box testing tools try to automate some of the techniques applied by penetration testers by using **pre-canned attacks**.
- Because these tools use close to the same set of attacks against every program, they are able to find only defects that do not require much meaningful interaction with the software being tested.
- Failing such a test is a sign of real trouble, but passing doesn’t mean very much; it’s easy to pass a set of pre-canned tests.

The Quality Fallacy

- Another approach to testing, **fuzzing**, *involves feeding the program randomly generated input* [Miller, 2007].
  - Testing with purely random input tends to trigger the same conditions in the program again and again, which is **inefficient**.
- To improve efficiency, a fuzzer should **skew** (*mutate*) the tests it generates based on knowledge about the program under test.
- If the fuzzer generates tests that resemble the file formats, protocols, or conventions used by the target program, it is more likely to put the program through its paces.
- Even with customization, fuzzing is a **time-consuming process**, and without proper iteration and refinement, the fuzzer is likely to spend most of its time exploring a **shallow portion** of the program's state space.

The Trinity of Trouble

- Why is making software behave so hard? Three factors work together to make software risk management a major challenge today.
- We call these factors the trinity of trouble:
  - **Complexity** - Modern software is complicated, and trends suggest that it will become even more complicated in future.
    - More Lines, More Bugs.
    - More configurations
    - More artifacts.
  - **Extensibility**
    - Malicious content can slip through extensions and affect core.
  - **Connectivity**
    - Malicious content can go from one place to another.