# COBOL Security Reviews for Fun and Profit

# Agenda

- whoami
- Introduction
- Overview of Mainframe Concepts
- Introduction to COBOL
- Carrying Out a COBOL Code Review
- Common Issues
- Common Non-Issues
- Reviewing with VCG (Yes! I have written a tool!)
- Reporting
- Questions

# whoami

@n1ckdunn.bsky.social
www.linkedin.com/in/npdunn

- Coming from software development and architecture
  - Started as a software developer, architect, team lead, working in secure software for the financial sector
  - Worked as an in-house penetration tester and code reviewer in online gambling
- Moved into security consultancy and worked on:
  - Code review
  - Penetration testing
  - Threat modelling, architecture review
  - Automating security testing with new tools, scripts, etc.
  - Security research
  - Author of upcoming book Black Hat R

# Introduction

Mainframes - why are they still here?

COBOL - why is it still here?

- Some large organisations, mostly financial sector, have had these systems in place for decades.
- Running large scale batch processing and amounts of data may be problematic for a less powerful machine.
- Removal of these systems is not easy and not cheap.

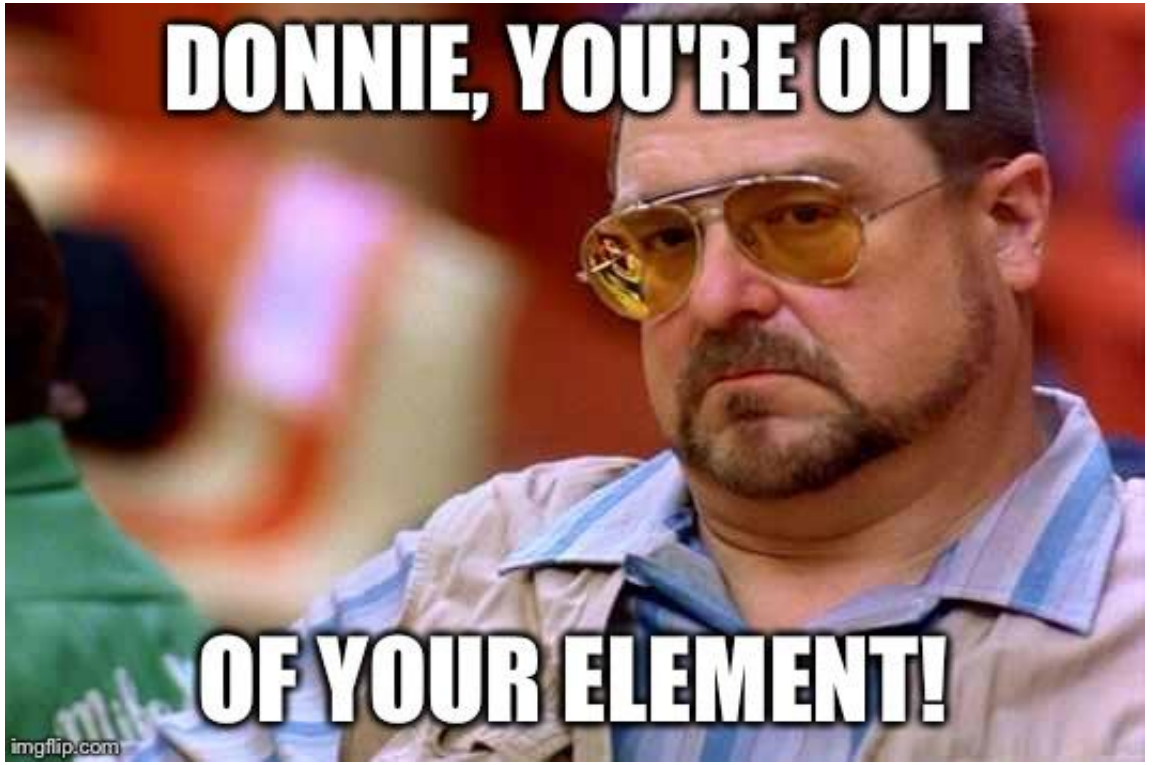
# Mainframe Concepts and Architecture

- More and faster processors
- More physical memory and greater memory addressing capability
- Enhanced devices for input/output with more and faster paths between devices and processors
- Fast and sophisticated inboard processing
- Great ability to divide resources of one machine into multiple, logically independent systems
- Advanced clustering technologies, and ability to share data among multiple systems

# What is COBOL?

You might expect it to be strange or obscure, but as we'll see, many of the concepts are familiar.

It can be fairly easily understood by anyone with some basic programming background.



# What is COBOL?

COBOL (Common Business Oriented Language) was devised in the 1950s by Admiral Grace Hopper.

It is the second oldest high-level language still in use – just slightly ‘newer’ than Fortran.

Originally devised as an easy to use programming language that would put an end to programming errors.

# What is COBOL?

COBOL was originally designed to allow non-mathematicians to program and as a result has some unusual syntax by the standards of other programming languages.

It can look unwieldy or weird:

ADD a TO b GIVING c.

As it became clear that using words instead of symbols did not make it magically possible for absolutely anyone to develop software, the language began to evolve:

a + b = c.

As COBOL was originally written by hand on specially formatted sheets of paper, code is formatted to fit the paper’s columns and follows a rigid format.

# Where are we Likely to See it?

It is still in use in many large financial organisations (banking, investment banking, insurance).

If you do review COBOL, you are likely to see code that has been written in the 1980s or earlier - core systems in use for decades still carry out heavy duty batch processing.

# Where are we Likely to See it?

Mainframes aren’t always old. Newer and more powerful systems are in place at some of the wealthier organisations.

# Reasons for a COBOL Code Review?

‘Never gone wrong before’ is not a guarantee that something is safe.

Some older code may be written under assumptions that no longer apply in modern environments.

There will be coloured text on a black background so the test will resemble Hollywood movie hacking.

# Preparing for a Code Review

There is a possibility that an administrator unfamiliar with the code security review process could initially provide you with just terminal access.

Attempt to find out ahead of time what their plans are and ensure that they don’t do this.

If you get the chance to prepare before the job, check out IBM's training materials.

There are multiple videos on YouTube and the IBM website. These have useful information and some comedy value.

# Structure of COBOL Programs

IDENTIFICATION DIVISION

ENVIRONMENT DIVISION

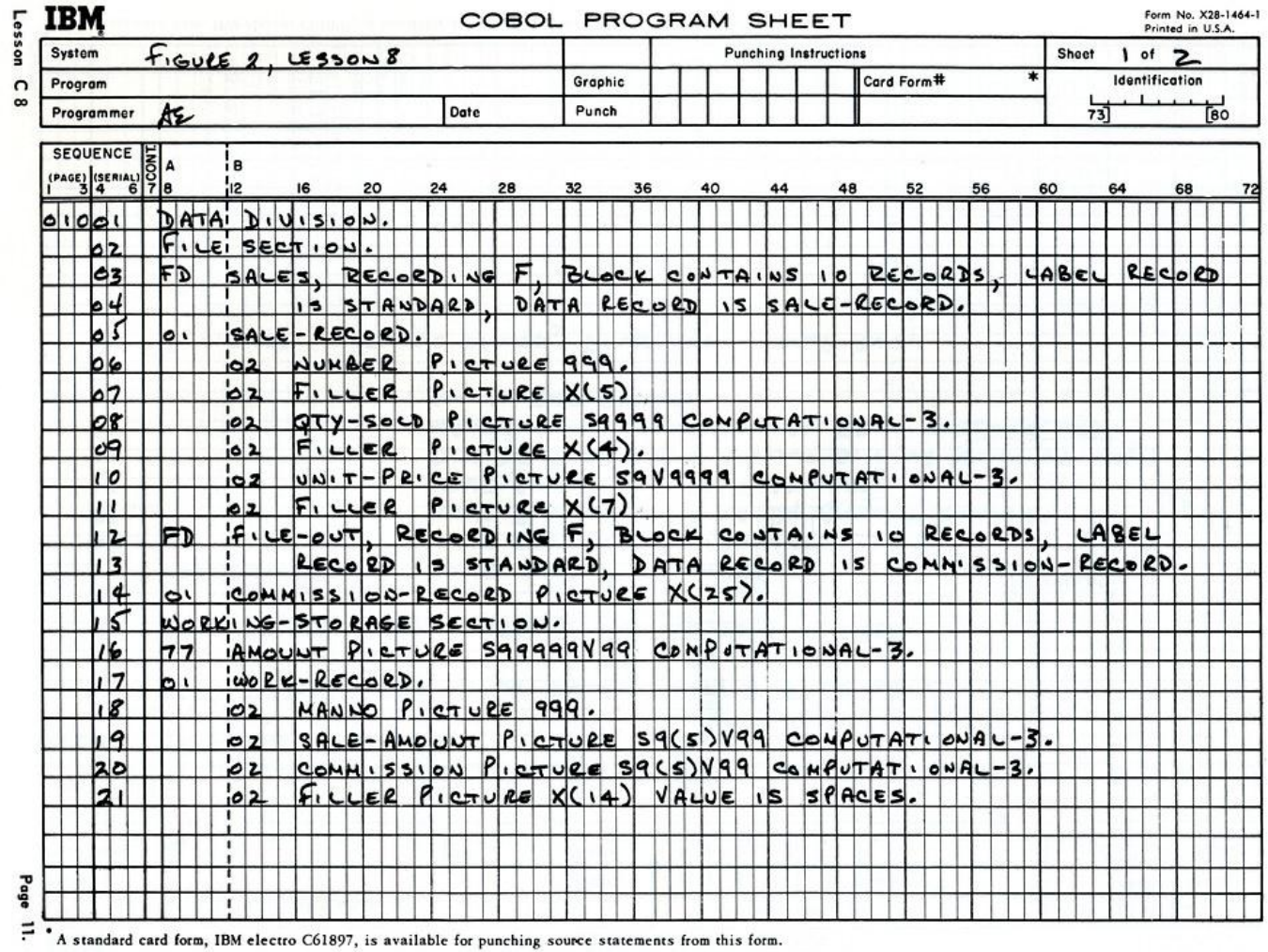
DATA DIVISION:

FILE SECTION, to define data used in input-output operations

LINKAGE SECTION, to describe data from another program.

PROCEDURE DIVISION

# Structure of COBOL Programs



Lesson C 8

IBM

COBOL PROGRAM SHEET

Form No. X28-1464-1
Printed in U.S.A.

System: FIGURE 2, LESSON 8 | Punching Instructions | Sheet 1 of 2

Program | Graphic | Card Form# * | Identification (73–80)

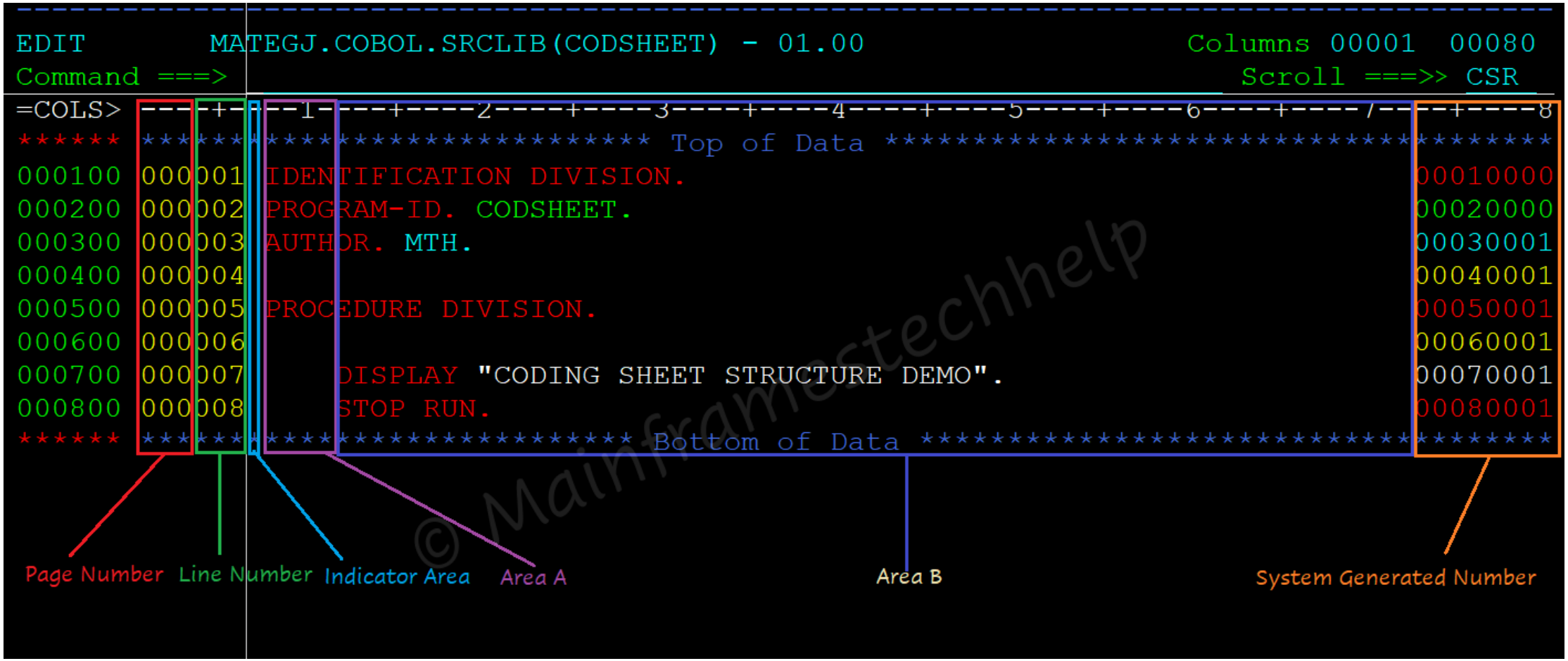
Programmer: AE | Date | Punch

| SEQUENCE (PAGE) 1–3 | (SERIAL) 4–6 | CONT 7 | A 8 | B 12–72 |
|---|---|---|---|---|
| 010 | 01 | | | DATA DIVISION. |
| | 02 | | | FILE SECTION. |
| | 03 | | | FD SALES, RECORDING F, BLOCK CONTAINS 10 RECORDS, LABEL RECORD |
| | 04 | | | IS STANDARD, DATA RECORD IS SALE-RECORD. |
| | 05 | | | 01 SALE-RECORD. |
| | 06 | | | 02 NUMBER PICTURE 999. |
| | 07 | | | 02 FILLER PICTURE X(5) |
| | 08 | | | 02 QTY-SOLD PICTURE S9999 COMPUTATIONAL-3. |
| | 09 | | | 02 FILLER PICTURE X(4). |
| | 10 | | | 02 UNIT-PRICE PICTURE S9V9999 COMPUTATIONAL-3. |
| | 11 | | | 02 FILLER PICTURE X(7) |
| | 12 | | | FD FILE-OUT, RECORDING F, BLOCK CONTAINS 10 RECORDS, LABEL |
| | 13 | | | RECORD IS STANDARD, DATA RECORD IS COMMISSION-RECORD. |
| | 14 | | | 01 COMMISSION-RECORD PICTURE X(25). |
| | 15 | | | WORKING-STORAGE SECTION. |
| | 16 | | | 77 AMOUNT PICTURE S99999V99 COMPUTATIONAL-3. |
| | 17 | | | 01 WORK-RECORD. |
| | 18 | | | 02 MANNO PICTURE 999. |
| | 19 | | | 02 SALE-AMOUNT PICTURE S9(5)V99 COMPUTATIONAL-3. |
| | 20 | | | 02 COMMISSION PICTURE S9(5)V99 COMPUTATIONAL-3. |
| | 21 | | | 02 FILLER PICTURE X(14) VALUE IS SPACES. |

Page 11.

* A standard card form, IBM electro C61897, is available for punching source statements from this form.

# Structure of COBOL Programs



# Carrying Out the Code Review

Where possible attempt to get files in some form where they can be directly transferred to your laptop so that you have some chance of searching for the issues below, using your choice of IDE, and whatever code review tools are available to you.

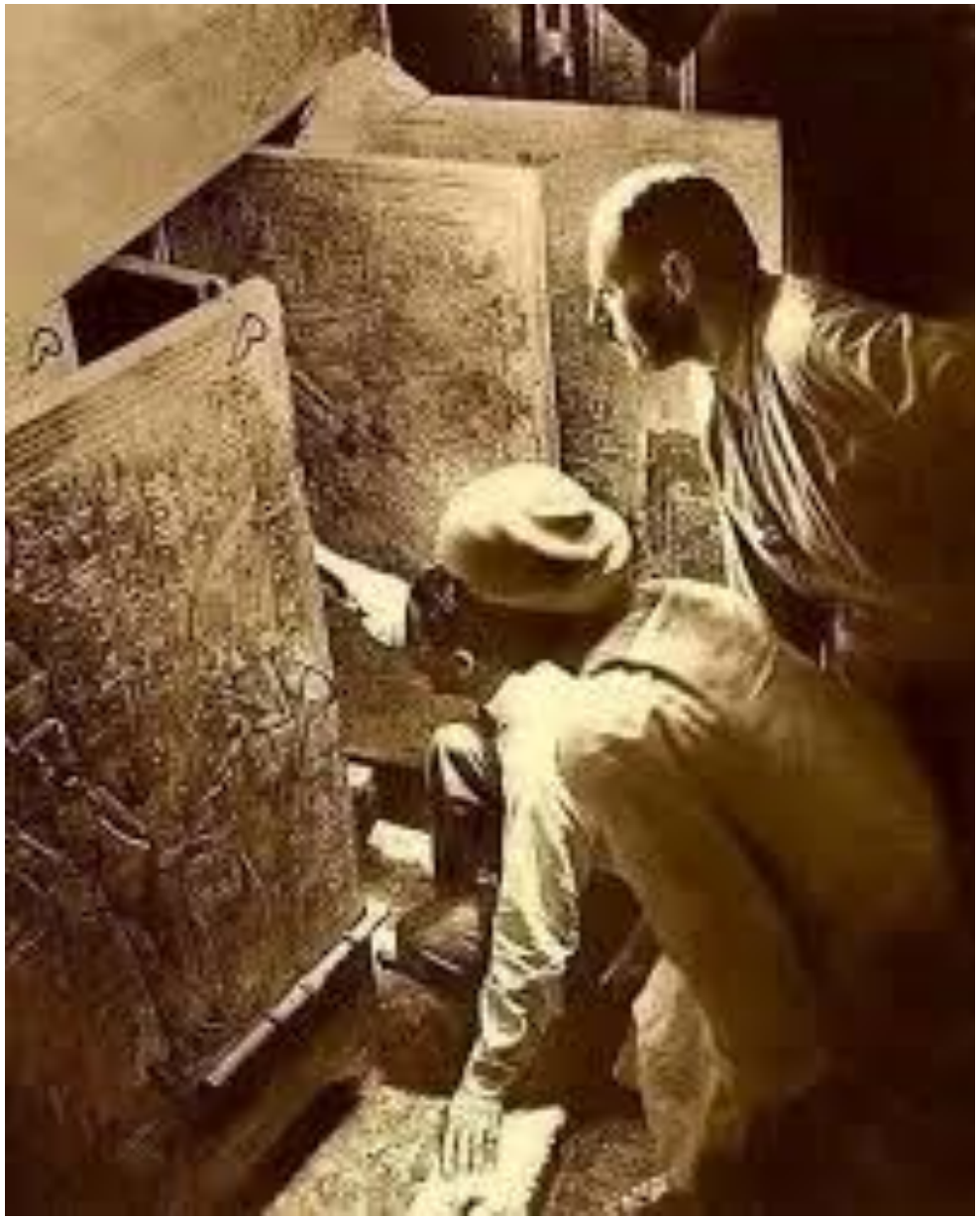
Obtain suitable tooling (VCG, CheckMarx, Veracode, others(?)) before the assignment.

# What to Look For

Some general concepts to be aware of:

- The nature of the systems means that we are dealing with slightly different problems than for web, mobile or desktop applications.
- Issues to report on generally fall into the categories of trusting the user (or the input) too much, or failing to anticipate future changes.

# How Do We Assess a System That's Worked Okay for 20 Years?



# Some Common Issues

The following fall into the ‘bad in the future, if things change’ category:

- Mismatched variable types in move operation
- Signed variable values moved to unsigned destination
- Mismatched lengths of pic variables used in move operations
- Trusting but not verifying input

# COBOL Variable Types

- The PIC variable type is our main concern when dealing with variable type, size, and sign mismatches:
  - PIC 9 (5) – numeric variable of size 5
  - PIC S9 (4) – signed numeric variable of size 4
  - PIC V (6) – implicit decimal variable of size 6
  - PIC P (6) – assumed decimal variable of size 6
  - PIC A (5) – alphabetic variable of size 5
  - PIC X (5) – alphanumeric variable of size 5

# Mismatched Variable Types in Move Operation

```
  +0100             02  CARD-START-POL                    PIC X(7).
...
  +0100         01  WS-START-POL                          PIC S9(7) VALUE 0.
...
  +0100               ELSE
  +0100                EVALUATE WS-SUB1
  +0100                  WHEN 1 MOVE CARD-ID             TO WS-JOBNAME
  +0100                         MOVE WS-ULP              TO WS-STREAM
  +0100                         MOVE WS-ULP-THF          TO WS-ULP-THF-R
  +0100                  WHEN 2 MOVE CARD-MAX-TIMESTAMP TO WS-MAX-TIMESTAMP
  +0100                                                    WS-CARD-MAX-TIMESTAMP
  +0100                  WHEN 3 MOVE CARD-START-POL      TO WS-START-POL
```

# Signed Variable Values Moved to Unsigned Destination

# Signed Variable Values Moved to Unsigned Destination

```
  +0127  056744          05  H-GAD-MAX-4                    PIC S9(3)V99 COMP-3
  +0127  056744                  VALUE  0.
...
  +0104  182971      03  WS24-GAD-MAX                       PIC 9(3)V99.
...
  +0104  182971              ELSE
  +0127  056744                   ADD 1 TO WS24-GAD-YY WS24-C2-YY
  +0127  056744                   IF WS24-COMP1-DATE < WS24-COMP2-DATE
  +0127  056744                       MOVE H-GAD-MAX-4 TO WS24-GAD-MAX
```

# Mismatched Lengths of PIC Variables Used in MOVE Operations

```
  +0111          01  WS35-TMP-SUB                              PIC 9(5)
...
  +0111          01  WS35-SUB                                  PIC S9(4) COMP
...
  +0124  029143      IF SOLAR-MOVT-NEW-FUND-NO (WS35-TMP-SUB) = ('000' OR SPACES)
  +0111                 OR WS35-TMP-SUB GREATER THAN WS35-TOTAL-OCC
  +0111                 MOVE WS35-TMP-SUB TO WS35-SUB
```

# Trusting External Data

The following code fragment illustrates the use of the ACCEPT command to read the input. The variables are populated from an external source, with the only validation being a check for empty input:

```
+0100                MOVE SPACES      TO   CARD-REC1
+0100                ACCEPT CARD-REC1 FROM SYSIN
+0100                IF  CARD-REC1 =  SPACES
+0100                    DISPLAY 'NO PARMS FOR ULE970'
+0100                    MOVE    'A010-GET-PARMS' TO WSTCSZ99-PARA-NAME
+0100                    PERFORM  ZROLLBACK
+0100                ELSE
+0100                 EVALUATE WS-SUB1
+0100                   WHEN 1 MOVE CARD-ID            TO WS-JOBNAME
+0100                          MOVE WS-ULP             TO WS-STREAM
+0100                          MOVE WS-ULP-THF         TO WS-ULP-THF-R
+0100                   WHEN 2 MOVE CARD-MAX-TIMESTAMP TO WS-MAX-TIMESTAMP
+0100                                                    WS-CARD-MAX-TIMESTAMP
+0100                   WHEN 3 MOVE CARD-START-POL     TO WS-START-POL
+0100                          MOVE CARD-END-POL       TO WS-END-POL
+0100                 END-EVALUATE
+0100                 ADD 1 TO WS-SUB1
+0100                END-IF
```

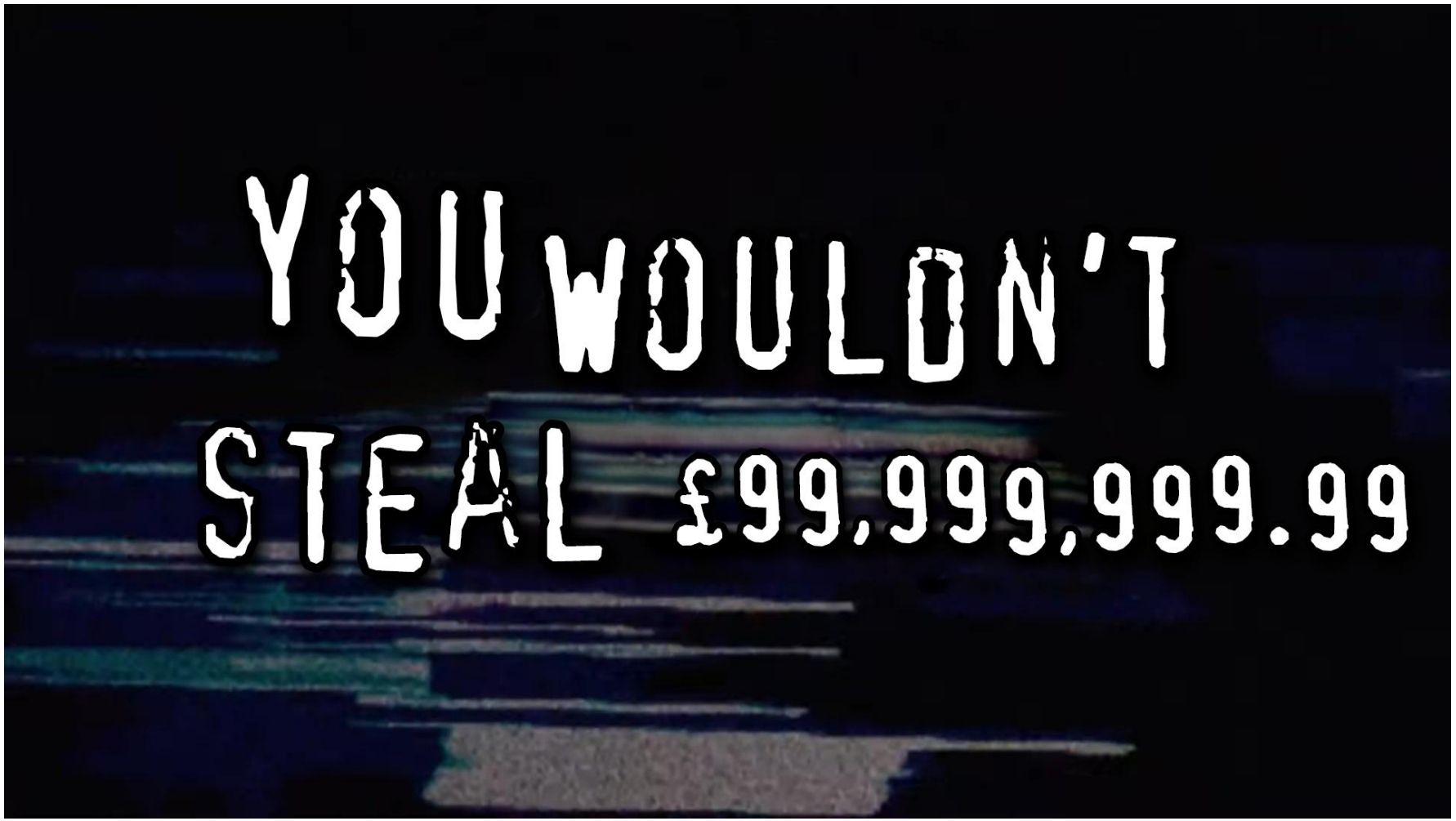
# SQL Injection in COBOL?!?!

You may have found SQL injection reviewing web code, and you might think of it is as an almost exclusively web-related issue.

COBOL allows construction of dynamic SQL statements that result in SQL injection.

One inadvertent defence is likely to be size of the input field, or in some cases the allowed characters of the input field.

# SQL Injection in COBOL

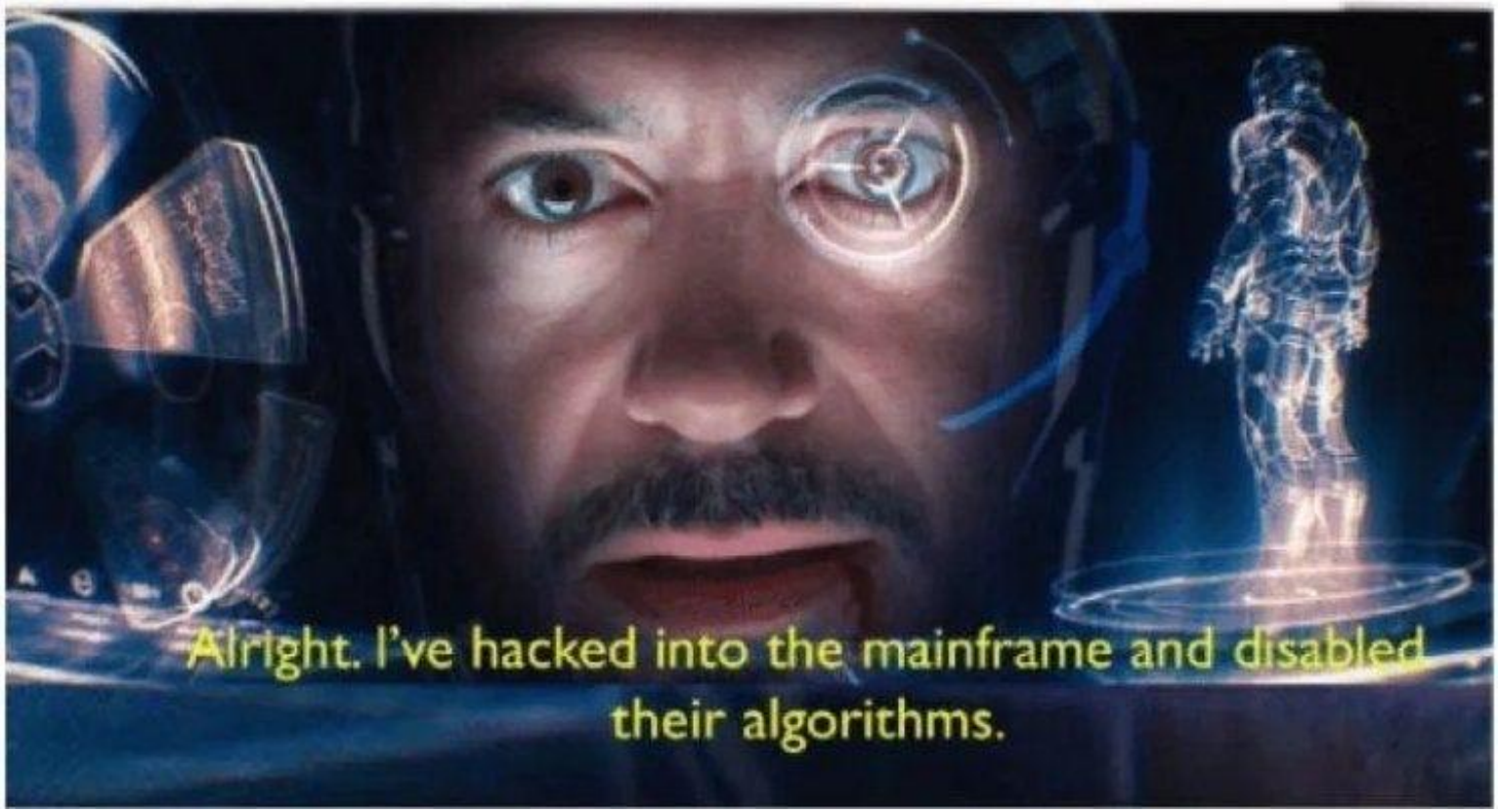


# COBOL SQL Injection Example

```
ACCEPT TRANS-CUST-ID

STRING ' SELECT CUST-ID INTO WS-CUST-ID'
    'FROM CUSTOMER'
    'WHERE CUST-ID = "'"  TRANS-CUST-ID "'"
    DELIMITED BY SIZE
  INTO WS-STR-TXT
END-STRING

MOVE LENGTH OF WS-STR-TXT TO WS-STR-LEN

EXEC SQL PREPARE ADDSTMT FROM :WS-STR    END-EXEC
EXEC SQL EXECUTE ADDSTMT  END-EXEC
```

# Buffer Overflows

- The idea of buffer overflows on a mainframe might make you think of ill-informed scenes in movies.

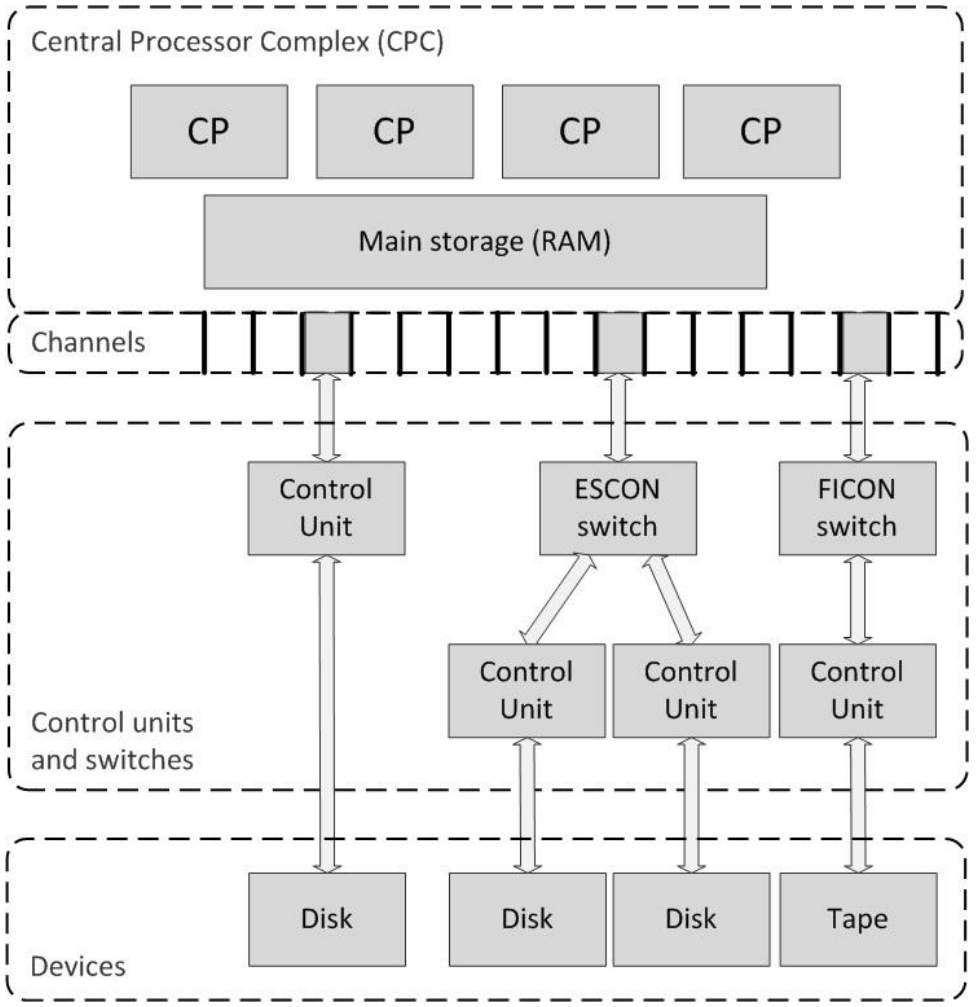


# Differences between COBOL and ‘Normal’ Languages

Architecture – the stack is implemented differently from an x86 machine, so overflows are less hazardous than in C (or C++) code.

COBOL provides an ON OVERFLOW phrase to defend against overflow errors, but an overflow will not cause the same catastrophic consequences that a C overflow would.

# Logically Complex Logical Architecture



# Functions to Look For

Like any language, COBOL has functions which are open to abuse or increase the likelihood of bugs.

The following functions should be flagged as indicators of limited code quality and may act as a starting point to find areas of potential trouble:

ALTER – In COBOL it's actually possible to write self-modifying code. The ALTER statement is used like this:

```
ALTER proc1 PROCEED TO proc2
```

Horrifyingly, this means that at compile time the code will look past the statement labelled as proc1 until it finds the next GOTO proc1 statement and dynamically patch this with a GOTO proc2 statement. This kind of twisted insanity makes the program harder to read, test, maintain and debug.

# More Functions to Look For

GOTO – This can be flagged as producing unmaintainable code with a greater likelihood of bugs. As you are likely to be looking at older programs which people are reluctant to modify because of both their age and a lack of problems in the past, you are highly likely to encounter ‘goto’ statements and just as likely to meet people who don’t want fix something that they perceive as not broken.

ENTRY POINT – This is COBOL’s statement label, used in conjunction with the ‘goto’ statement.

File Management – Check for safe file management and validation of untrusted data. This may be common in older applications due to the trusted nature of the environment in the past. The DISPLAY and OPEN verbs are used for file management so search for these and check for any excessive trust in the input data.

# Tooling

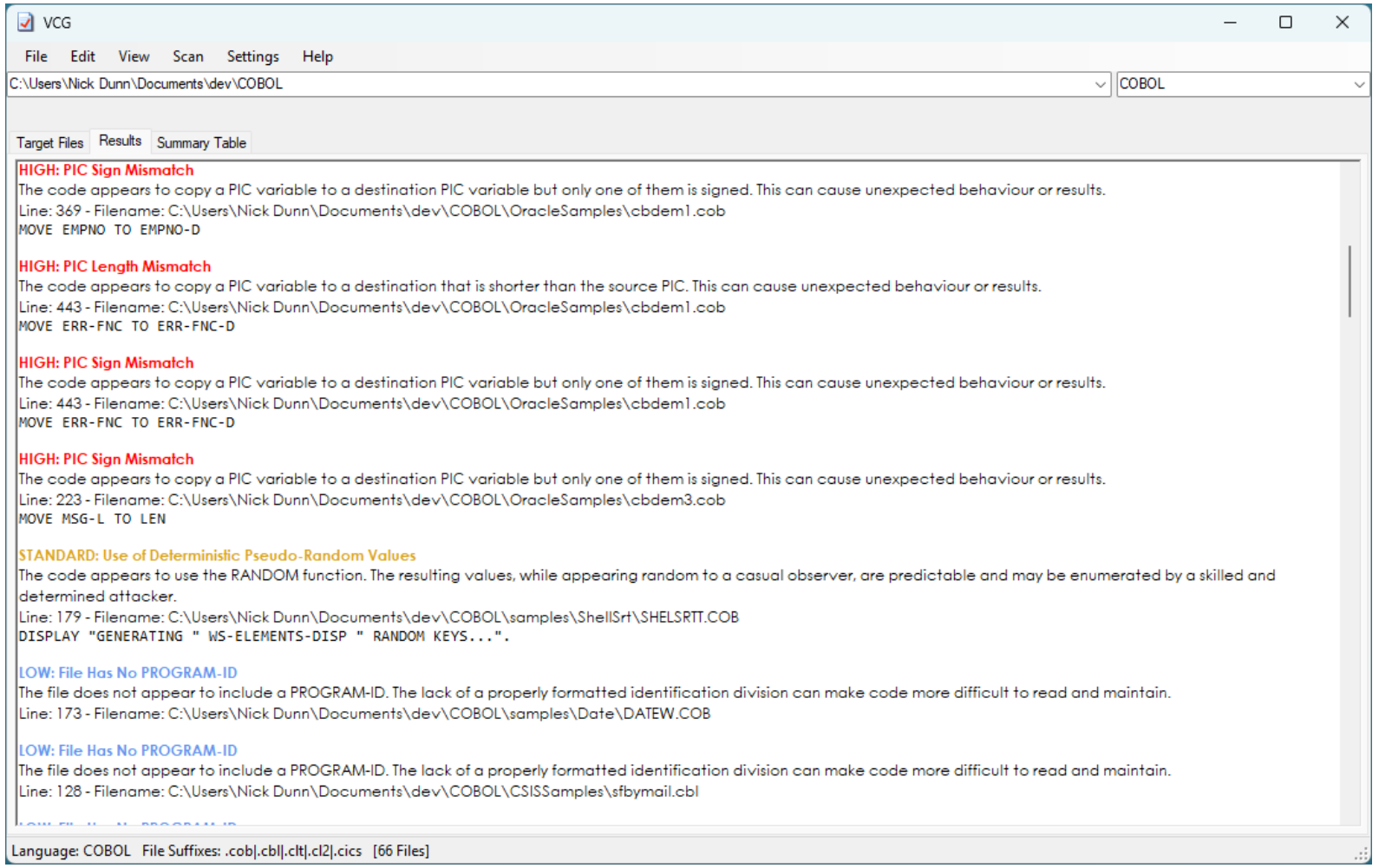
Veracode, CheckMarx and other commercial tools can scan COBOL code.

Latest version of VCG is a free tool that can scan COBOL:

https://github.com/nccgroup/VCG

https://sourceforge.net/projects/visualcodegrepp/

# Typical VCG Output



# Reporting Issues and Making Fixes

# Maintenance and Good Practice

A comprehensive report helps to emphasize that the job has been carried out correctly and methodically. The following are not security issues, but you should report back if they're not being followed.

COBOL-specific points to look for regarding code quality and ease of maintenance:

- It's helpful for the PROGRAM-ID to match the source file's name, particularly for a large or complex application.
- A source file should not contain more than one PROGRAM-ID.
- It's helpful if different programs are located in different files.
- Ideally, the executable module(s) should be placed in the "execute only" library to prevent them from being read by unauthorised users.

# Reporting and Explaining Results

The applications have likely been running for years with no issues, and so there will be a need for context for any findings.

e.g. If a numeric field is being copied to a smaller numeric field this may have gone unnoticed because the amounts of money (it will almost always involve money) have been shorter than the field length.

This is not future-proof and may cause a serious issue if/when the amounts are larger.

# Reporting and Explaining Results

Mainframes have traditionally been protected by physical means and access in the distant past took place using a ‘dumb terminal’.

Some explanation should be given to explain how user-controlled data can be dangerous for a mainframe that is available on an internal network.

# Conclusions

The language is not as weird or obscure as you may have thought.

Applications that have run safely for years can still have issues that might be a problem in the future.

There may be issues (such as SQLi) that are very serious but just haven’t been detected.

Green text on a black background lets you experience the same sort of 'hacking' that you see in movies.

# Any Questions?

# Thank You!