# Download Tools From SMB

- Please download tools at \\CyberEvent\resources\BinaryDecompilation\Software
  - Copy everything under ./Ubuntu to your local machine if you are using Ubuntu
  - Copy everything under .\Windows to your local machine if you are using Windows
  - It mostly does not matter what OS you use. You can reboot your laptop if you want to switch OS.

# Binary Decompilation

CyberTruck Challenge 2023

Fish Wang

# Who am I?

- Professor at Arizona State University (ASU)
  - Publish papers on top-tier security venues
  - Build open-source tools (e.g., angr)
  - Enjoy debugging programs
- Core member of *Shellphish* 

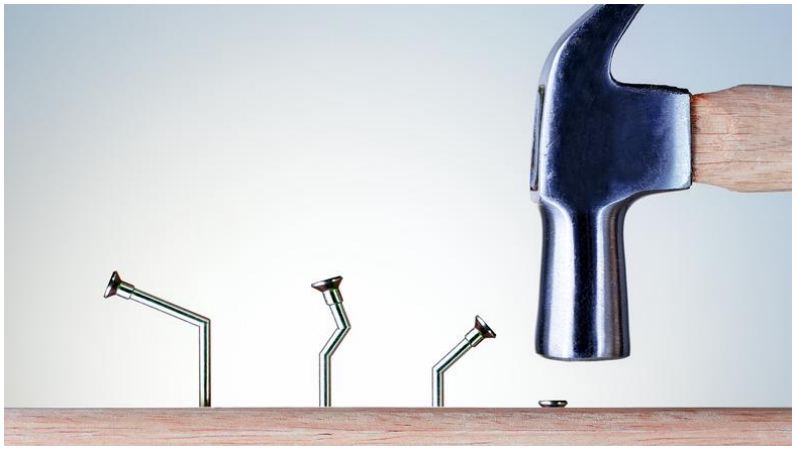
  - Longest running team at DEF CON CTF
- Member of *Nautilus Institute*
  - Running DEF CON CTF since 2022

# How about Decompilation?

- My dream since I was eight
  - I spent countless hours with OllyDbg
  - Did not know IDA Pro (nor did it have a working decompiler back then)
- Developed angr decompiler mostly by myself
  - Standing on giants' shoulders!
- Actively working on decompilation research
  - Variable name inference
  - Better structuring algorithms
  - AI-powered quality-of-life improvements

# Format

- Lecture
  - Basic concepts
  - How things work
  - Research directions
- Expert suggestions
  - What to decompile?
  - How to do it fast?
- Hands-on challenges
  - Various architectures
  - They are easy!



# TOOLS

# Pick Your Poison

- For this event, you should **only use locally installable tools**
  - No cloud versions
- Try as many decompilers as you can
  - When one does not work, switch to another
- See README.md for a list of recommended decompilers

# Download Tools From SMB

- Please download tools at \\CyberEvent\resources\BinaryDecompilation\Software
  - Copy everything under ./Ubuntu to your local machine if you are using Ubuntu
  - Copy everything under .\Windows to your local machine if you are using Windows
  - It mostly does not matter what OS you use. You can reboot your laptop if you want to switch OS.

# IDA Pro

- IDA Pro is the state-of-the-art tool for reversing
  - https://www.hex-rays.com/products/ida/
- It supports disassembling of binary programs
- Supports decompilation (Hex-Rays decompiler)
- Can be integrated with GDB and other debuggers
- It is a commercial product (expensive)
  - A limited version is available for free

# Binary Ninja

- Disassembler that supports sophisticated analysis
  - https://binary.ninja/
- Includes multiple intermediate representations
  - Easier to read than pure assembly code!
- It is a commercial product but:
  - It has a demo version that can be used for free (with limitations)
  - It is not very expensive (149$)
  - It is way more stable than other competitors

# Ghidra

- Open-source disassembler from NSA
  - https://ghidra-sre.org/
  - https://github.com/NationalSecurityAgency/ghidra
- Supports multiple architectures
- Supports decompilation (side-by-side view)

# angr

- angr is an open-source binary analysis platform
  - Disassembling
  - Lifting to multiple IRs
  - Symbolic execution
- Great support for Python scripting
- Has a GUI
  - angr management
- Free
- Much easier to use

# Decompiler Explorer

- There are a few other decompilers. How do we pick one (or several) that we want to use?

- https://dogbolt.org

# Lab -1. BYOD

- Download and install various decompilers. Ensure they run on your laptop.

# Lab 0. Let’s Decompile

- Let’s start from some simple binaries
  - Find the input to each binary that leads to “Congrats!”

# Step 0: What to do?

- The very first task is understanding your goal
  - Finding flags?
  - Finding passwords?
  - Negating a condition?
  - Bypassing a protection?

# Step 1: How is it implemented?

- Finding beacons
  - *Beacons* are recognizable **patterns** during reverse engineering
  - Usually beacons are human-readable strings
  - They can also be
    - Common C constructs (switch-case)
    - API calls

# Step 2. Understanding Logic

- What logic is implemented in the program?
  - Top-down
    - Understand the logic by reading source code line after line, starting from the *main* function or the entry point of the binary
  - Bottom-up
    - Understand the logic by reading relevant source code *around* the beacons you care about
  - Making the decompiled code more readable as you go
    - Rename variables
    - Rename functions
    - Add comments
    - Retype variables, functions, and function calls

# Step 3. Reflection

- Reflection is extremely important during reverse engineering
  - "How would I implement this feature if I were the programmer?"
  - "How would I protect this function if I were the programmer?"
  - "What mistakes would I make if I were the programmer?"
  - It's all about experience

# WHY DECOMPILATION?

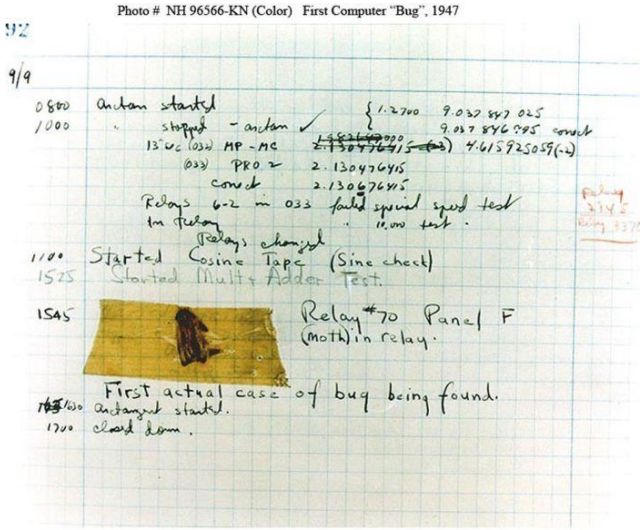
# Why Decompilation?

- Debugging
  - Things have gone wrong… but what and why?
- Understanding
  - What is the logic? What is implemented?
- Editing


- Reusing
  - Use the code/function/module/binary elsewhere

# Debugging

- Admiral Grace Hopper (1906 – 1992)
  - Her associates found a moth stuck in a relay of Mark II and removed it.
  - Grace Hopper first used the term *debugging* in the context of computing

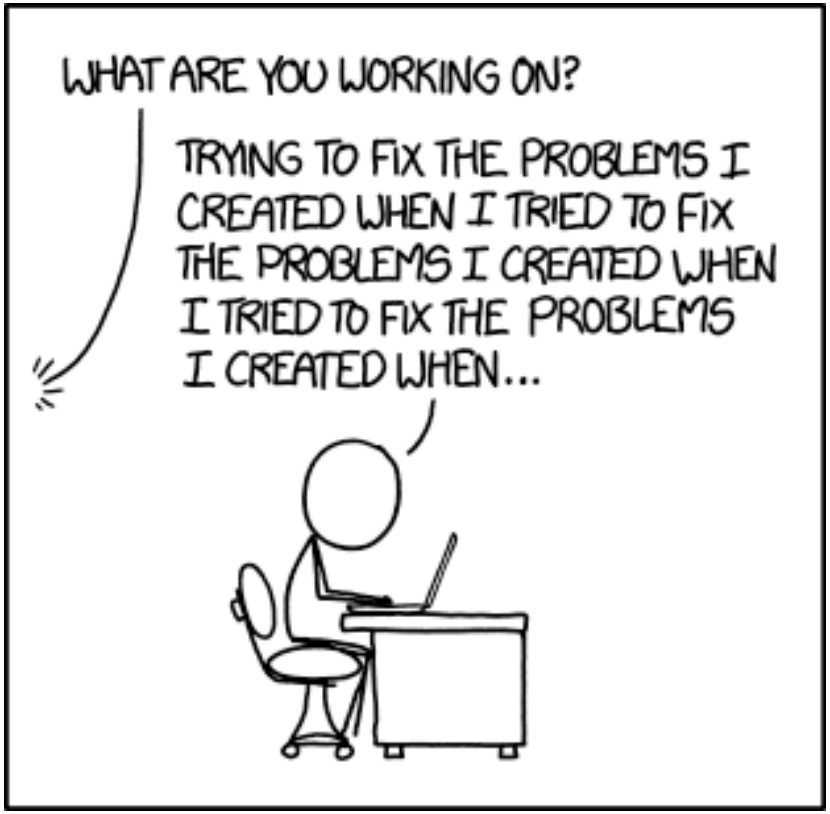


# Modern Debugging

- Finding (root) causes of errors, bugs, and failures in computing systems
  - Manually
  - Automatically

- Debugging is difficult because software is unnatural, complex, and sometimes functions in unexpected ways

# You are part of the problem!




https://xkcd.com/1739/

# Software Bugs are Prevalent

- “Commercial software typically has 20 to 30 bugs for every 1,000 lines of code…”
  - Wait what? One bug per 50 lines of code?
- What can we do to reduce the number of bugs?
- The fewer lines of code, the better
  - “Code is liability” – Dr. Yan Shoshitaishvili

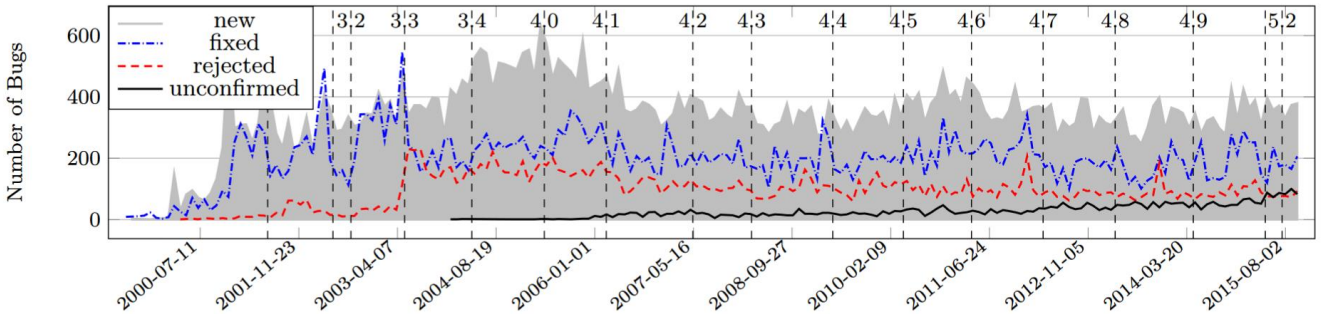
# You are not the only problem

- Writing less code does not necessarily make your software less complex
  - Language interpreters
    - CPython interpreter
    - Java VM
  - Language Runtime
    - Java Runtime
    - .Net CLR
  - Libraries
    - Glibc/uClibc/dietlibc/msvcrt/…
    - Java libraries
    - Python standard libraries
  - What else?

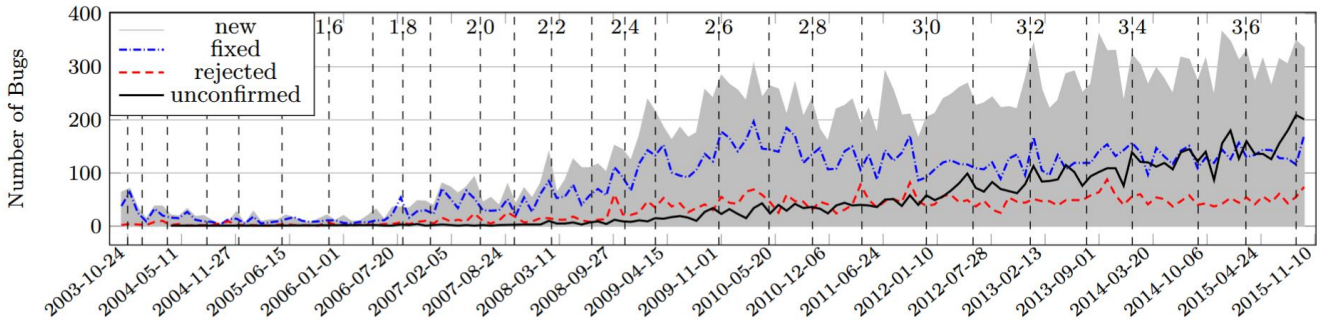
# Compiler Bugs

- Writing less code does not necessarily make your software less complex
  - Compilers
    - GCC/Clang
    - G++/Clang++
    - Go
    - Rust
    - Crystal
    - Julia
    - Ocaml
    - javac (JIT)

# Are Compiler Bugs Rare?




(a) GCC.




(b) LLVM

Figure 1: The overall evolution history of the bug repositories (in months). The plot filled in gray background shows the new bug reports submitted every month. The blue *dashdotted* plot shows the number of bugs fixed every month. The red *dashed* plot shows the number of bugs that are resolved as *not* fixed (*e.g.* invalid, duplicate, worksforme or wontfix). The black curve shows the number of bug reports per month which have not been confirmed yet. Clearly there is an increasing trend of unconfirmed bugs for LLVM.

# GCC 3.2 Bug Fixes

- https://gcc.gnu.org/gcc-3.2/changes.html

**Internal Compiler Errors (multi-platform)**

- 3782: (c++) `-quiet -fstats` produces a segmentation fault in cc1plus
- 6440: (c++) template specializations cause ICE
- 7050: (c++) ICE on: `(i ? get_string() : throw)`
- 7741: ICE on conflicting types (`make_decl_rtl` in `varasm.c`)
- 7982: (c++) ICE due to infinite recursion (using STL set)
- 8068: exceedingly high (infinite) memory usage
- 8178: ICE with `__builtin_ffs`
- 8396: ICE in `copy_to_mode_reg`, in `explow.c`
- 8674: (c++) ICE in `cp_expr_size`, in `cp/cp-lang.c`
- 9768: ICE when optimizing inline code at `-O2`
- 9798: (c++) Infinite recursion (segfault) in `cp/decl.c:push_using_directive` with recursive `using` directives
- 9799: mismatching structure initializer with nested flexible array member: ICE
- 9928: ICE on duplicate `enum` declaration
- 10114: ICE in `mem_loc_descriptor`, in `dwarf2out.c` (affects sparc, alpha)
- 10352: ICE in `find_reloads_toplev`
- 10336: ICE with `-Wunreachable-code`

**C/optimizer bugs:**

- 8224: Incorrect joining of signed and unsigned division
- 8613: `-O2` produces wrong code with builtin `strlen` and postincrements
- 8828: gcc reports some code is unreachable when it is not
- 9226: GCSE breaking argument passing
- 9853: miscompilation of non-constant structure initializer
- 9797: C99-style `struct` initializers are miscompiled
- 9967: Some standard C function calls should not be replaced when optimizing for size
- 10116: ce2: invalid merge of `join_bb` in the context of `switch` statements
- 10171: wrong code for inlined function
- 10175: `-Wunreachable-code` doesn't work for single lines

# VC6 Variable Scope Leaks

```
for (int myvar = 0; myvar < 10; myvar++);
if (1)
{
    int var2 = 16;
}
myvar = 0;
var2 = 0;
```

https://stackoverflow.com/questions/4075276/variable-scope-in-c

# Seemingly Compiler Bugs

- Null checks in C

```
if (ptr != NULL) {
    // it's a valid pointer!
    do_something(ptr);
}
```

  - Accessing null pointers is considered an undefined behavior in C
  - Some C compilers introduced an optimization -fdelete-null-pointer-checks
  - `if (ptr != NULL)`
    `error("Invalid driver");`

# Seemingly Compiler Bugs

- Null pointer checks are silently removed without programmers realizing it…
- Worse, null pointers are actually dereferenceable on some platforms (mostly embedded)
  - Linux needs to run on many platforms
  - Hence https://bugzilla.redhat.com/show_bug.cgi?id=511185
  - It was proposed in 2009!

# Debugging & Reverse Engineering

- Debugging
  - “What is wrong with my code?”
- Reverse engineering
  - “What is wrong with this program?”
  - “What is wrong with this binary?”
  - Maybe it was my own code from six months ago…
- Decompilers make it much easier to understand binary code (machine code)

# DECOMPILATION 101

# Disassembling

- Disassembling is the process of extracting the assembly representation of a program by analyzing its binary representation

- Disassemblers can be
  - Linear
    Linearly parse the instructions
  - Recursive
    Attempt to follow the execution flow of the program

# Static Analysis

- Static analysis is a technique to analyze programs that does not involve executing the program
- Control-flow analysis
  - Analyzes how the program execution is transferred across the program components
    - CFG: Control-flow graph
- Data-flow analysis
  - Analyzes what data values can be assumed by specific data stores (e.g., variables) at various points in the program

# Manual Decompilation

- Convert each instruction into its equivalent C representation
  - Do other languages work?
- Manually perform common compiler optimizations
- Identify variables and their types
- Infer variable semantics
- Infer function semantics

# Wait… Architectures? Platforms?

- Theoretically, a decompiler converts raw binary code into a high-level representation
  - So you don't need to understand architecture-specific or platform-specific details
- In reality…
  - Decompilers frequently make mistakes
  - Architecture- and platform-specific nuances are often preserved
  - Decompilers do not always work

# Lab 1. A Challenge with Crypto

- csaw2012reversing.exe
  - Find the correct decrypted flag.

# Lab 2. A Bit More Crypto

- regme
  - Find a correct flag that this binary takes.

# What's inside a decompiler?

- Many decompilers are open-sourced

- angr: angr/analyses/decompiler/decompiler.py
- Reko: src/Decompiler/Decompiler.cs
- Snowman: nc/core/MasterAnalyzer.cpp

# Decompilation Techniques

- Control-flow graph recovery
  - Differentiating between code and data
- (Optionally) Lift to a higher-level intermediate representation
- Optimize the representation to eliminate redundant accesses (e.g., register accesses)
- Recognize and replace compiler-specific idioms
- Control-flow structure analysis
  - Recovering the high-level control-flow structures
- Variable recovery and type inferencing
- Pretty-printing of the representation into C

# Lab 3. More Reversing

- ais3_crackme
  - Find the correct flag that this binary takes.

# Lab 4. Self-modifying Binary

- bonnie
  - Find the correct flag that this binary takes.

# Library Functions

- You don't implement everything on your own. Instead, we *stand on the shoulders of giants*
- Who are giants in the world of software engineering? Libraries

- Library functions implement logic that can be reused
  - C: libc, pthread, math, libcrypto, …
  - Python: re, logging, hashlib, …
  - Qt: …

# Understanding Library Calls

- Library calls have well-defined interfaces and functionality. You do not need to guess what they do!
  - Man pages
  - Library documentation
  - GitHub
  - Google

# Invoking System Calls

- System calls are usually invoked through libraries
  - `libc`
- Linux/x86_64
  - `syscall`
    - `rax` contains the system call number

- What syscalls are there? Is there a reference?
  - Here! https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/

# System Calls are OS-specific

- Platform-specific Linux syscall tables

# Lab 5. Binary Patching

- pacman
  - Identify the main function that implements the game logic (the function with a large switch-case struct).
  - The ghosts are moving too fast. Make them move slower!
  - Make pacman immortal!