Level 0x06

Binary Ninja

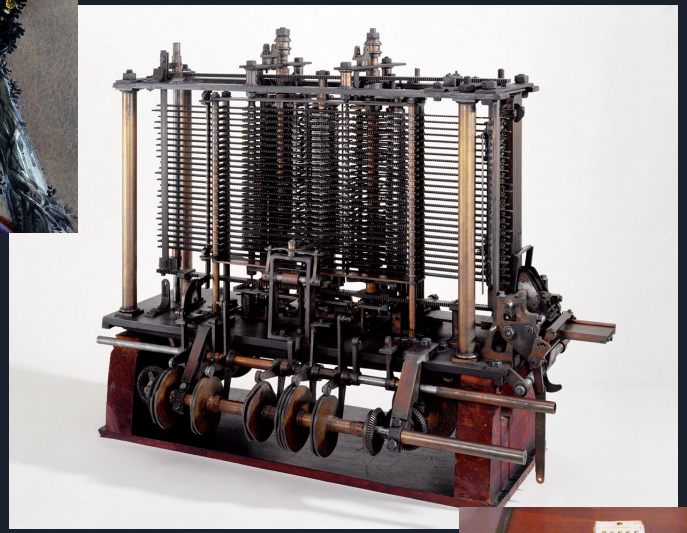
Topics

- Events
- RE Tools

Advent of Code

- Created by Eric Wastl in 2015
- Got huge during Covid / no place to go, let's code!
- What is it?
  - Coding challenge each day of December until Dec 25th
  - They start out easier and get more difficult
- Good competitive programming practice (there is a top 100 leaderbo
- Professionals participate too
  - Learn a new programming language
  - Visualized solutions
  - Bragging rights
  - Weird / fun solutions
- Check out the subreddit

Ada Lovelace



- English mathematician in UK 1815 - 1852
- Worked with Charles Babbage on Analytical Engine
  - Babbage never finishes unit #1 (pic on right)
  - In 1991 a London Museum built unit #2
  - Could do 4 types of math ops, branch, and loop
  - Had memory
  - Programmed with cards, data on cards too
- Ada Byron Lovelace
  - translates articles about the analytical engine
  - made some notes about how a program would look like to calculate Bernoulli numbers
  - this is considered the first computer program
- Ada programming language
- IDA Pro logo

Reverse Engineering Tools

IDA Pro (1991)

- Written by HexRays / Ilfak Guilfanov
- Free / $365 / >$7K
- ~ 40 CPUs / 11 C
- Belgium / Russia

Ghidra (2003 / 2019)

- Written by NSA
- Mostly free (some is still TS)
- 30 CPUs supported
- Java / Slow…

Binary Ninja (2016)

- Written by Vector35
- 10 CPUs
- C++ / Qt / Fast
- Python scripting
- Free / $300 / $3K

Binary Ninja GUI

Disassembly Level (very helpful!)

Different views of variables

Buttons to change side dock contents

game.bndb — Binary Ninja Personal 4.1.4967-dev

File Edit View Analysis Debugger Plugins Window Help

game.bndb

Symbols | Search symbols | ELF ▾ | Linear ▾ | Pseudo C ▾

Name	Adc
_init	0x0
sub_8049030	0x0
__libc_start_main	0x0
sub_8049046	0x0
printf	0x0
sub_8049056	0x0
fflush	0x0
sub_8049066	0x0
getchar	0x0
sub_8049076	0x0
fgets	0x0
sub_8049086	0x0
signal	0x0
sub_8049096	0x0
sleep	0x0
sub_80490a6	0x0
__stack_chk_fail	0x0
sub_80490b6	0x0
puts	0x0
sub_80490c6	0x0
exit	0x0
sub_80490d6	0x0
fopen	0x0
sub_80490e6	0x0
putchar	0x0
sub_80490f6	0x0
_start	0x0
sub_804912d	0x0

Mini Graph

```
int32_t main(int32_t argc, char** argv, char** envp)

08049764  int32_t main(int32_t argc, char** argv, char** envp)

08049764  {
0804976b      void* const __return_addr_1 = __return_addr;
08049772      int32_t* var_10 = &argc;
08049784      void* gsbase;
08049784      int32_t eax = *(uint32_t*)((char*)gsbase + 0x14);
08049796      struct PlayerData player;
08049796      init_player(&player);
080497ac      struct MapData mapData;
080497ac      updateMap(&mapData, &player);
080497c5      printMapAndStatus(&mapData, &player);
080497d9      signal(2, exitAndDontReturn);
08049805      while (true)
08049805      {
08049805          movePlayer(&player, getchar(), &mapData);
0804981e          printMapAndStatus(&mapData, &player);
0804982f          if (player.yCoordinate == 0x1d)
0804982f          {
0804983a              if (player.xCoordinate == 0x59)
0804983a              {
0804983a                  break;
0804983a              }
0804982f          }
08049805      }
08049846      puts("You win!");
08049857      if (player.hasFlag != 0)
08049857      {
08049863          puts("flage");
0804986b          printsFlag();
0804987c          fflush(*(uint32_t*)stdout);
08049857      }
0804988d      *(uint32_t*)((char*)gsbase + 0x14);
08049894      if (eax == *(uint32_t*)((char*)gsbase + 0x14))
08049894      {
080498a4          return 0;
08049894      }
08049896      __stack_chk_fail_local();
08049896      /* no return */
08049764  }

080498a5  void* const sub_80498a5() __pure
```

Stack

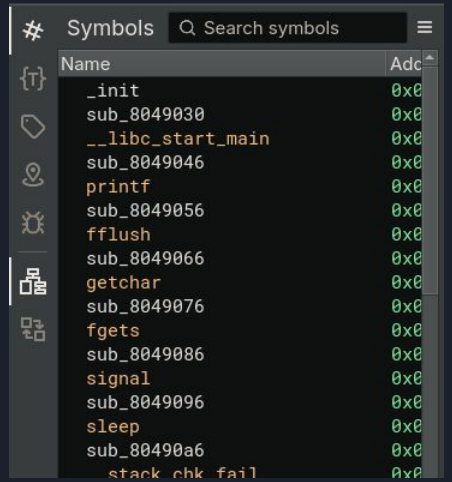
```
-0xac0   MapData* var_ac0
-0xac0   struct PlayerData* var_ac0_1
-0xac0   struct MapData* var_ac0_2
-0xac0   FILE* var_ac0_3
-0xabc   struct PlayerData* var_abc
-0xabc   char var_abc_1
-0xabc   struct PlayerData* var_abc_2
-0xab8   struct MapData* var_ab8
-0xab8   struct MapData* var_ab8_1
-0xab4   struct PlayerData* var_ab4
-0xab4   struct PlayerData* var_ab4_1
-0xab0   void var_ab0
-0xab0   ?? ?? ??
-0xaad   char var_aad
-0xaad   char var_aad_1
-0xaac   struct PlayerData player {
-0xaac     int32_t yCoordinate
-0xaa8     int32_t xCoordinate
-0xaa4     char hasFlag
-0xaa3     ?? ?? ??
-0xaa0   }
-0xaa0   struct MapData mapData {
-0xaa0     char positions[0x1e][0x5a]
-0x014   }
-0x014   int32_t var_14
-0x010   int32_t* var_10
-0x00c   int32_t __saved_ebx
-0x008   int32_t __saved_ebp
-0x004   void* const __return_addr_1
 0x000   void* const __return_addr
 0x004   int32_t argc
 0x008   char** argv
 0x00c   char** envp
```

12 | linux-x86 | 0x8049805–0x804980a (0x5 bytes)

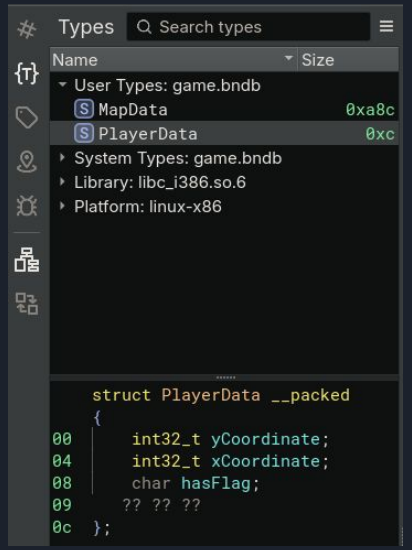
Side Dock Modes

Symbols:
- Functions
- Global Variables

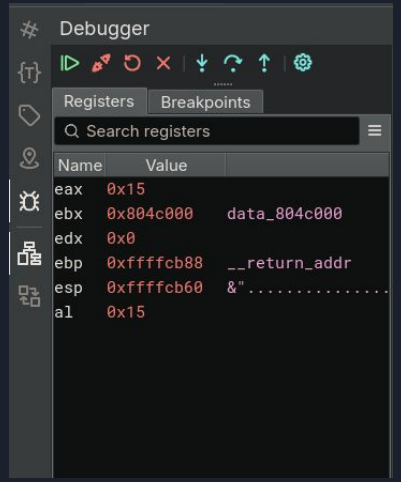
You can filter / search list



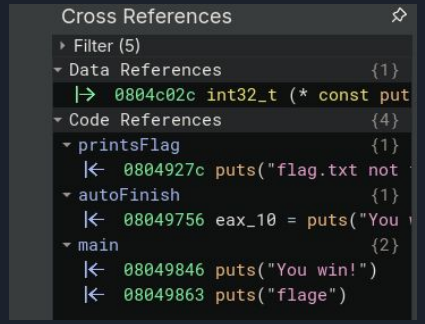
Structures / Types



Debugger
- Low level / you see register vals



Cross References
- Who calls this function?
- Who uses this variable?



Main Window

Main part of the UI for Binary Ninja can show different views

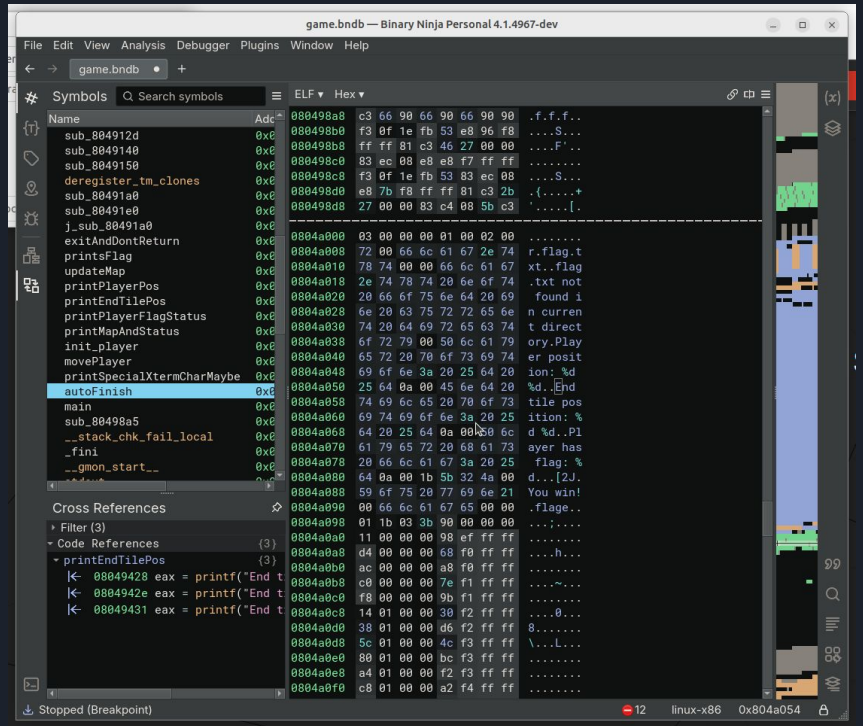
- Strings (just like strings command, but now we can click on them / see where used)
- Hex view (are these bytes code, a string, a number, any patterns?)
- Linear view (looks like a code listing / source code)
- Graph view (looks like a flow chart, can see loops / branches)

Code itself has different views

- Disassembly / Machine language (changes based on architecture)
- Intermediate Levels (LLIL, MLIL, HLIL. pseudo code)
- Language Level (C, Rust, Python)

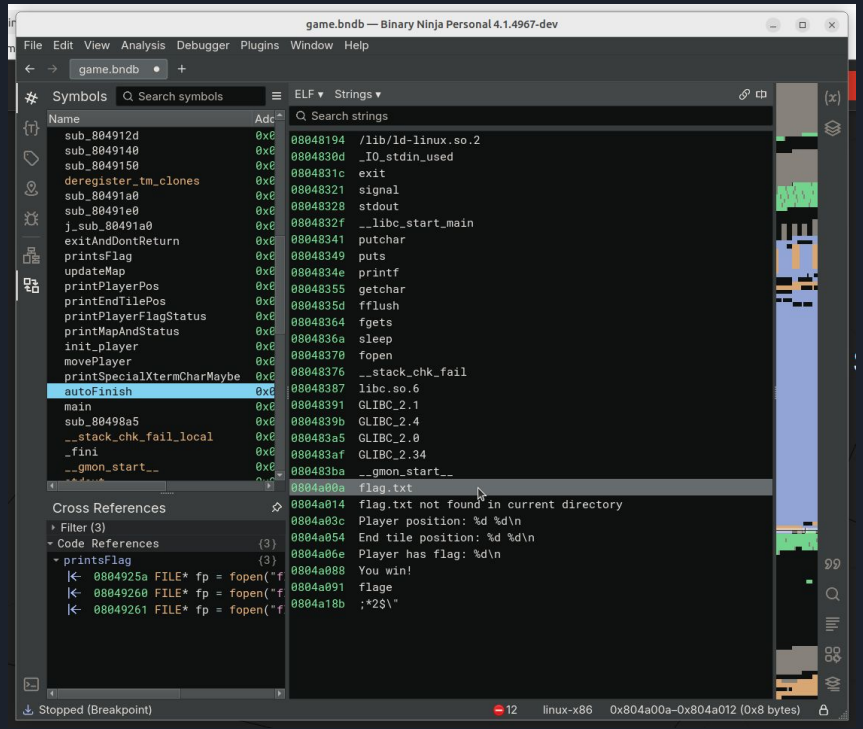
Hex View

- Address where data is
  - Not file offset
  - Address within ELF section
- Cross-refs show what code / data refers to the data



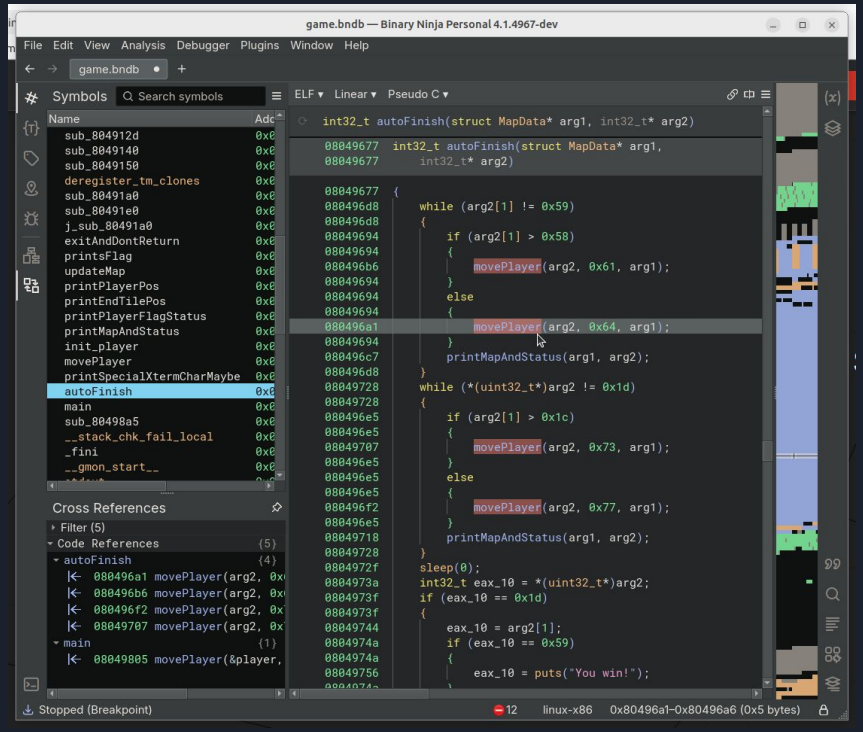
Strings View

- Address where string is
- String itself
  - A string is series of printable chars
  - Has a 0-byte / null terminator
- Cross-refs show what code / data refers to the string



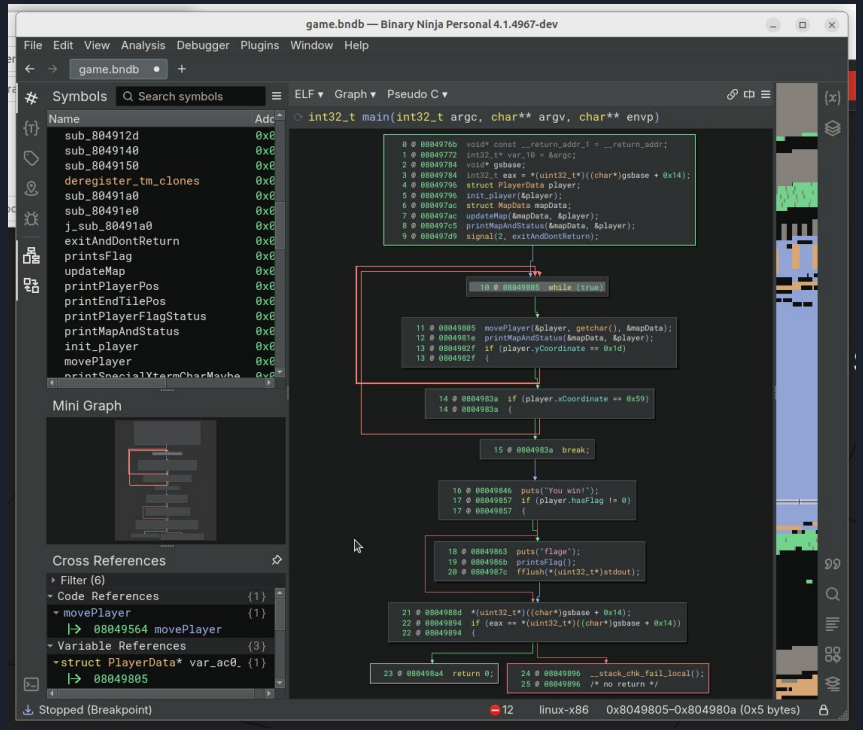
Linear View

- Code listing
- Branches have C style indented blocks
- You can toggle display of numeric values
  - Hex or decimal
  - String / printable ASCII
- Looks like code that we write



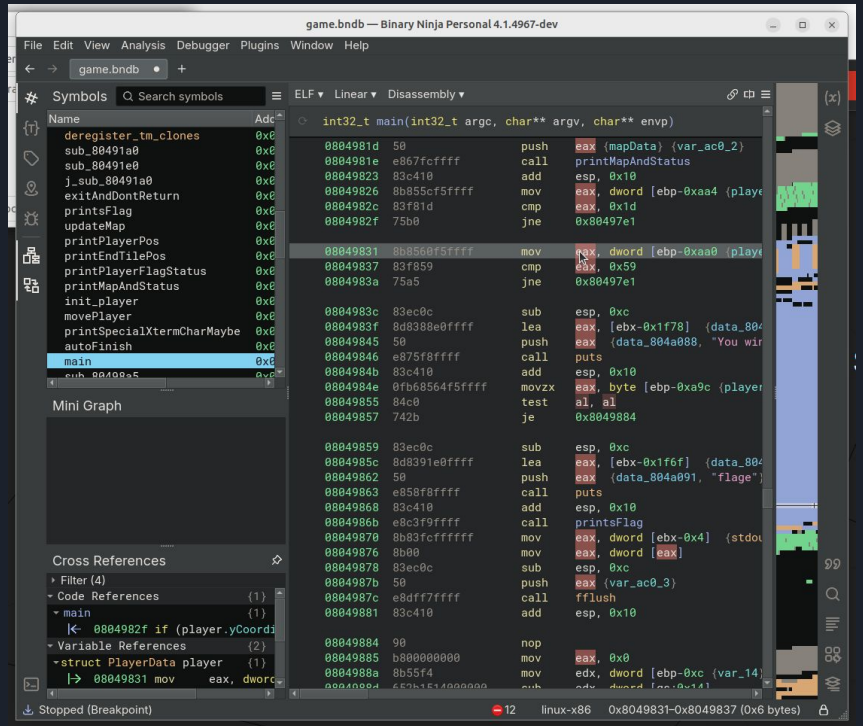
Graph View

- Blocks of code in graph nodes
- Can zoom in / zoom out
- Sometimes this view makes more sense for complicated code flows
- Mini-graph view on left side helps / has a fully zoomed navigation
- Probably more useful when looking as disassembly that high-level language



Disassembly View

- Shows instructions for the processor
  - You have to understand the assembly syntax for CPU you are reversing
- Op codes
  - Notice how not all instructions are the same length on x86
- Registers (and the variable they represent)
- Assembly code branching very hard to understand in this in linear view



IL Views

- Low-Level IL is like assembly for generic CPU type
- High-Level IL is like high level language, but still very descriptive about what instructions are doing

ELF ▾ Linear ▾ Low Level IL ▾

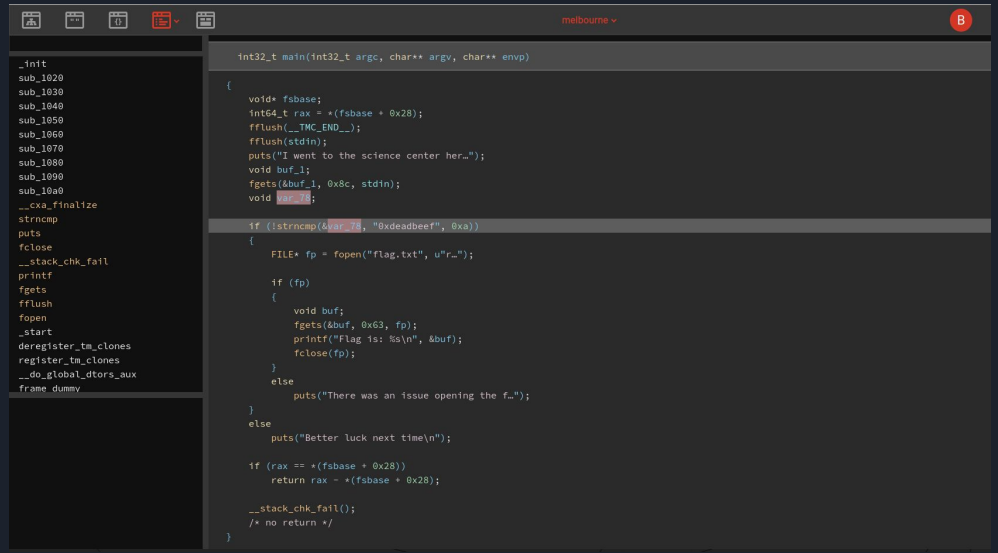
```
int32_t main(int32_t argc, char** argv, char** envp)

49 @ 08049810  eax = ebp - 0xaa4 {player}
50 @ 08049816  push(eax)
51 @ 08049817  eax = ebp - 0xa98 {mapData}
52 @ 0804981d  push(eax)
53 @ 0804981e  call(printMapAndStatus)
54 @ 08049823  esp = esp + 0x10
55 @ 08049826  eax = [ebp - 0xaa4 {player.yCoordinate}].d
56 @ 0804982f  if (eax != 0x1d) then 37 @ 0x80497e1 else 57 @ 0

57 @ 08049831  eax = [ebp - 0xaa0 {player.xCoordinate}].d
58 @ 0804983a  if (eax != 0x59) then 37 @ 0x80497e1 else 59 @ 0

59 @ 0804983c  esp = esp - 0xc
60 @ 0804983f  eax = ebx - 0x1f78
61 @ 08049845  push(eax)
62 @ 08049846  call(puts)
63 @ 0804984b  esp = esp + 0x10
64 @ 0804984e  eax = zx.d([ebp - 0xa9c {player.hasFlag}].b)
65 @ 08049857  if (al == 0) then 66 @ 0x8049885 else 72 @ 0x804

66 @ 08049885  eax = 0
67 @ 0804988a  edx = [ebp - 0xc {var_14}].d
68 @ 0804988d  temp0.d = edx
69 @ 0804988d  temp1.d = [gsbase + 0x14].d
70 @ 0804988d  edx = edx - [gsbase + 0x14].d
71 @ 08049894  if (temp0.d == temp1.d) then 85 @ 0x804989b else

72 @ 08049859  esp = esp - 0xc
73 @ 0804985c  eax = ebx - 0x1f6f
74 @ 08049862  push(eax)
75 @ 08049863  call(puts)
76 @ 08049868  esp = esp + 0x10
77 @ 0804986b  call(printsFlag)
78 @ 08049870  eax = [ebx - 4 {stdout}].d
79 @ 08049876  eax = [eax].d
80 @ 08049878  esp = esp - 0xc
```

ELF ▾ Linear ▾ High Level IL ▾

```
int32_t autoFinish(struct MapData* arg1, int32_t* arg2)

08049756            eax_10 = puts(str: "You win!")
08049763        return eax_10

08049764  int32_t main(int32_t argc, char** argv, char** envp)

0804976b        void* const __return_addr_1 = __return_addr
08049772        int32_t* var_10 = &argc
08049784        void* gsbase
08049784        int32_t eax = *(gsbase + 0x14)
08049796        struct PlayerData player
08049796        init_player(&player)
080497ac        struct MapData mapData
080497ac        updateMap(mapData: &mapData, player: &player)
080497c5        printMapAndStatus(mapData: &mapData, &player)
080497d9        signal(sig: 2, handler: exitAndDontReturn)
08049805        while (true)
08049805            movePlayer(player: &player, keyboardInput: getc
0804981e            printMapAndStatus(mapData: &mapData, &player)
0804982f            if (player.yCoordinate == 0x1d)
0804983a                if (player.xCoordinate == 0x59)
0804983a                    break
08049846        puts(str: "You win!")
08049857        if (player.hasFlag != 0)
08049863            puts(str: "flage")
0804986b            printsFlag()
0804987c            fflush(fp: *stdout)
0804988d        *(gsbase + 0x14)
08049894        if (eax == *(gsbase + 0x14))
080498a4            return 0
08049896        __stack_chk_fail_local()
08049896        noreturn

080498a5  void* const sub_80498a5() __pure
```

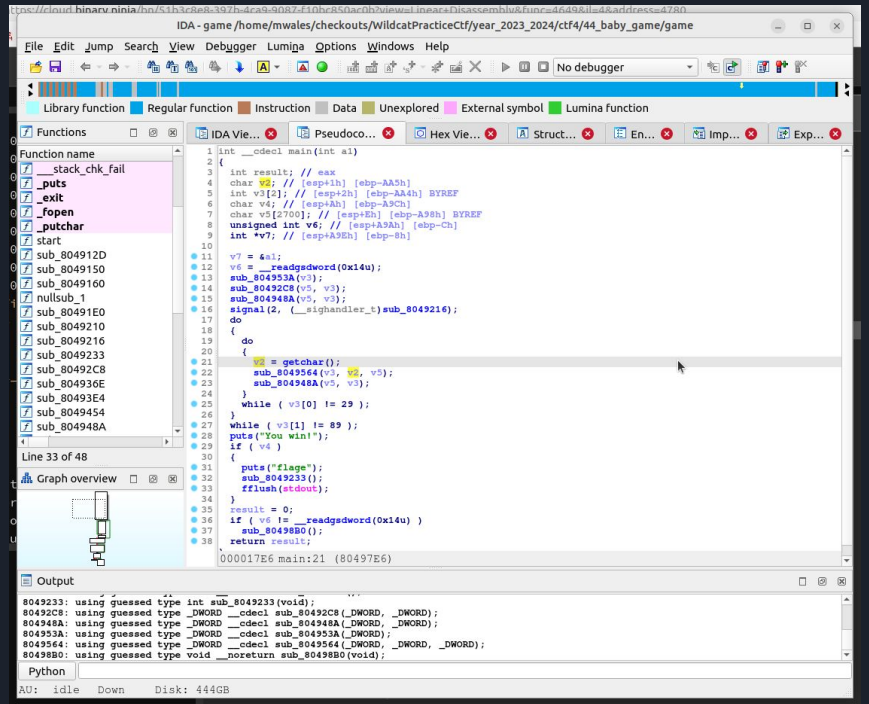
Free Cloud Version

- Register at cloud.binary.ninja
- Same underlying technology
- Missing many of the important features



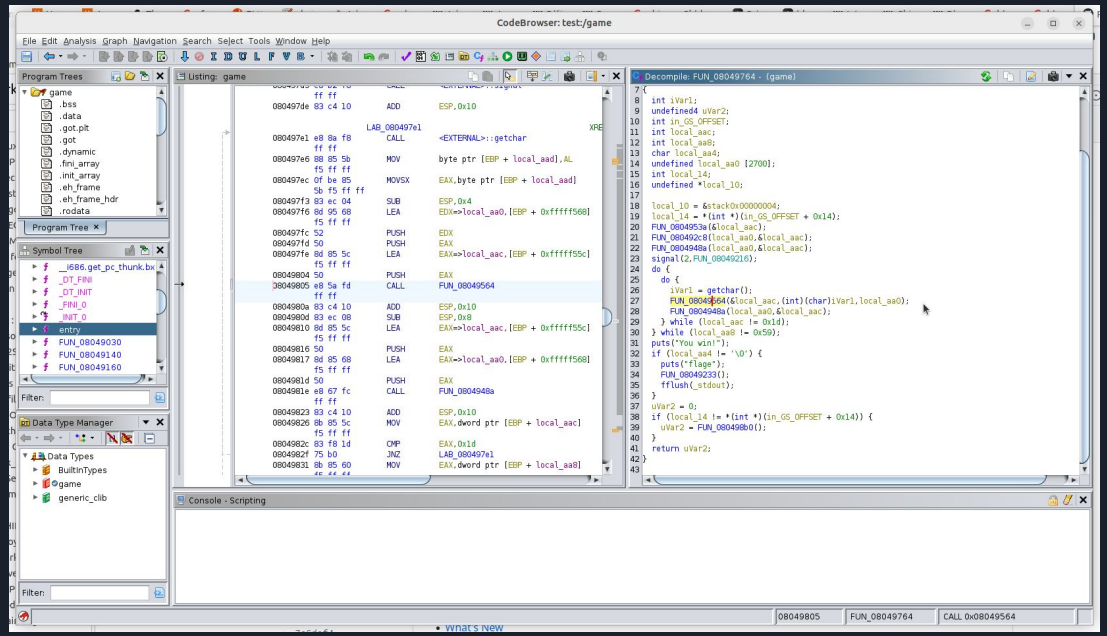
IDA Pro

- Industry standard tool / oldest
- Dissassembly vs decompilations
  - Decompiler was add-on feature
  - Seperate application in many ways
  - Names of variables don't always propagate between the views
  - Not all CPUs have decompilers
  - Decompilers are expensive!
  - X86 decompiler considered the best
- Free version for individuals
  - Cloud decompilation



Ghidra

- Free
- Harder to setup (project based)
- Side-by-side view of disassembly and decompiled code
- Supports teamwork
  - Server to store data
  - Sharing of structures
- Java based
  - Works on any OS
  - Slow on all of them



What are we trying to do?

- Identify structures (usually need to figure out the size of the structure)
  - Things that look like offsets from pointer / arrays
  - Group of related variables passed by pointer to functions
- Name functions, variables, structure fields
  - Guess what the variables should be called based on how code uses them
  - Shortcut key n : rename
- Set variable types
  - Shortcut key y : set type
  - Shortcut key d : rotate integer types
  - Shortcut key u : undefine
- Repeat
  - Shortcut key ESC : go back to previous location

Trying to get Binary Ninja disassembly / decompilation to look more similar to what the original source would look like so we can easily understand it

Links

-