ERNW
Living Security.

The Art of Reversing
A structured Approach

Michael Thumann

#whoami

- **Head of Research & Chief Security Officer, ERNW GmbH**

- **Recent Talks and Publications:**
  - “Hacking SecondLife”, Hack-in-the-Box, Dubai 2008
  - “Reversing – A structured approach”, RSA, San Francisco 2008
  - “Hacking Second Life”, Blackhat, Amsterdam, 2008
  - “Hacking the Cisco NAC Framework”, Sector, Toronto, 2007
  - “Hacking SecondLife”, Daycon, Dayton 2007
  - “Hacking Cisco NAC”, Hack-in-the-Box, Kuala Lumpur, 2007
  - “NAC@ACK”, Blackhat-USA, Las Vegas, 2007
  - “NAC@ACK”, Blackhat-Europe, Amsterdam, 2007
  - “Mehr IT-Sicherheits durch PenTests”, Book published by Vieweg 2005

- **What I like to do**
  - Breaking things ;-) and all that hacker stuff
  - Diving (you would be surprised what IT-Security lessons you can learn from diving)

- **Contact Details:**
  - Email: mthumann@ernw.de / Web: http://www.ernw.de

#whois ERNW GmbH

ERNW Living Security.

- **Founded in 2001**
- **Based in Heidelberg, Germany (+ small office in Lisbon, PT)**
- **Network Consulting with a dedicated focus on InfoSec**
- **Current force level: 18 employees**
- **Key fields of activity:**
  - Audit/Penetration-Testing
  - Risk-Evaluation & -Management, Security Management
  - Security Research
- **Our customers: banks, federal agencies, internet providers/ carriers, large enterprises**

Agenda

- **Part 1 – Introduction (very short)**
  - Why Reverse Engineering and why a structured approach
- **Part 2 – Needed Know How**
  - All you need to know in order to do it :-)
- **Part 3 – Tools of the Trade**
  - The Toolset – tools used at ERNW
- **Part 4 – The structured Approach**
  - How to make life more easy
- **Part 5 – Exercises**
  - Time to wake up guys

Part 1 - Introduction

Reverse Engineering Ninjitsu

- Not many people can do it
- Ninjas are invisible and can appear and disappear at any time
- Ninjas are all magicians !
- Ninjas are the bad guys
- But many people would like to know all that magic
- You can’t learn it from books, because the magic is not in the books

Reverse Engineering Ninjitsu - demystified

- **It’s not magic**
- **It’s all about Knowledge**
- **It’s all about the right techniques**
- **It’s all about the right weapons**
- **And it’s all about the right combination of knowledge, techniques and weapons**

Reverse Engineering - Definition

- is the process of discovering the technological principles of a device or object or system through the analysis of its structure and functions. It often involves taking something (mechanical device, electronic component, software program) apart and analyzing its workings in detail, usually to try to make a new device or program that does the same thing without copying anything from the original.

Why Reversing?

- **Because you need to know how the stuff is working**
- **Because Applications are very often distributed as binaries only**
- **Because a customer wants you to answer the question "Is this application secure?"**
- **Because finding security flaws is pretty cool and makes a good reputation for you and your company**
- **… and there are much more reasons ;-)**

Why structured

- Because Reversing all stuff needs to much time
- Because time is money ;-)
- Because the customer doesn't want to pay us for years to answer his question
- Because you won't get a result when you get lost in tons of code

Part 2 – Needed Know How

Needed Know

- Processor Architecture (RISC vs. CISC, Little vs. Big Endian and so forth)
- Assembler (there's more than one dialect ;-))
- OS internals
- OS API
- Commonly used programming languages
- Debugging
- Tool usage
- … and sometimes the ability to think in a way other people don't

Part 3 – Tools of the Trade

Needed Tools

- Disassembler
- Decompiler
- API Monitor
- Debugger
- Code Coverage Tools
- Sniffer
- Documentation ☺
- Your brain ☺

Commercial Must Have Tools

- **Disassembler: IDA Pro Advanced**
- **Decompiler: Hex-Rays (IDA Plugin)**
- **API Monitor: Autodebug Professional**

IDA Pro

- The famous and allmighty Disassembler
- Available for Windows, Linux and Mac OS X
- Commercial Product ($515 to $985)
- Debugger included that also supports debugging of PDAs
- Programmable and extensible (SDK included)
- Moved from Datarescue to Hex-Rays at the beginning of 2008
- Further Information at www.hex-rays.com

Hex-Rays Decompiler

ERNW Living Security.

- First Decompiler that produces more than crap
- Build by Ilfak Guilfanov (think IDAPro ☺)
- Released as commercial Addon for IDA (ca. $2.000)
- Planned: API to support Decompiler Plugins like Vulnerability Analyzer and others (First SDK Beta already released)
- Planned: Type and Function Prototype Recovery
- Planned: Assembler Knowledge not needed anymore
- Further Information at www.hex-rays.com

Autodebug API Monitor

- Debugger and API Monitor
- Watch the function calls and see the parameters passed to the function
- Commercial Tool ($299)
- Remote Debugging using a debug agent
- Used in our Cisco NAC Research and saved so much time
- Further Information at www.autodebug.com

Free Tools

ERNW Living Security.

- **Debugger: OllyDBG (www.ollydbg.de)**
- **Debugger: Immunity Debugger (www.immunitysec.com)**
- **Sniffer: Wireshark (www.wireshark.net)**
- **Decompiler: Boomerang (boomerang.sourceforge.net), free, but the output is more or less useless**
- **Code Coverage: PAIMEI (pedram.openrce.org/PAIMEI)**
- **Others: Log files ;-))**

More Commercial Tools

- HBGary Inspector: Cool AllinOne Tool, but more pricy (7K Bucks), but also worth a look
- Zynamics BinNavi: Flowcharts and Code Coverage (about 5k bucks)

Part 4 – The structured Approach

The ugly stuff –a structured approach

- Step 1: Define the question to answer
- Step 2: Understand the program flow
- Step 3: Identify interesting functions
- Step 4: Figure out the function prototype (used parameters)
- Step 5: Understand what the function is doing
- Step 6: Do runtime analysis to understand what the program is doing with input and output data
- Step 7: Use the gained knowledge to answer the question from Step 1

Tools used for the different steps

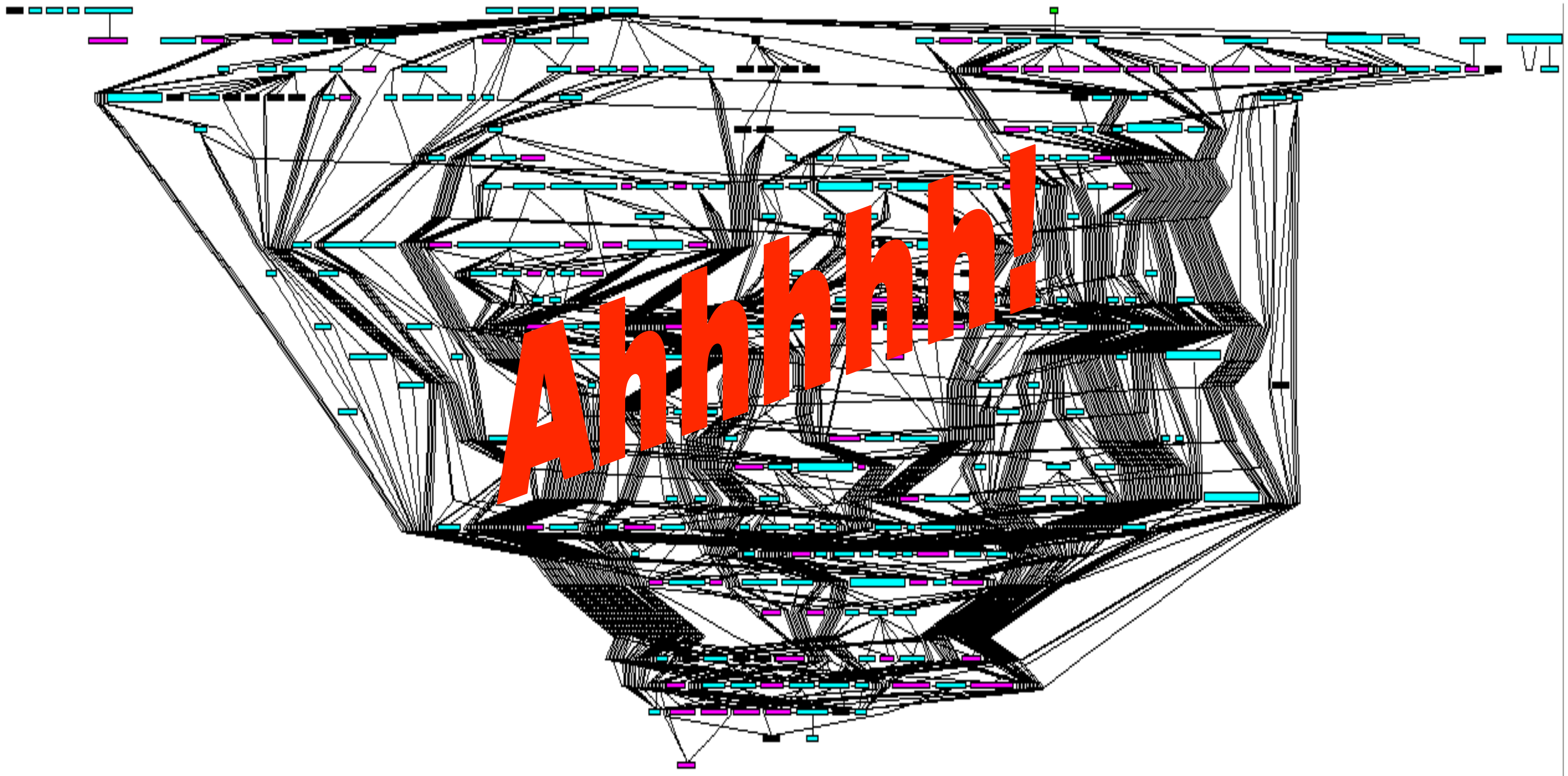
- **Step 1: The Brain V1.0**
- **Step 2: IDAPro**
- **Step 3: IDAPro**
- **Step 4: IDAPro / Hex-Rays**
- **Step 5: IDAPro / Hex-Rays**
- **Step 6: Autodebug / OllyDBG / Immunity Debugger**
- **Step 7: The Brain V2.0**

Time to wake up ☺

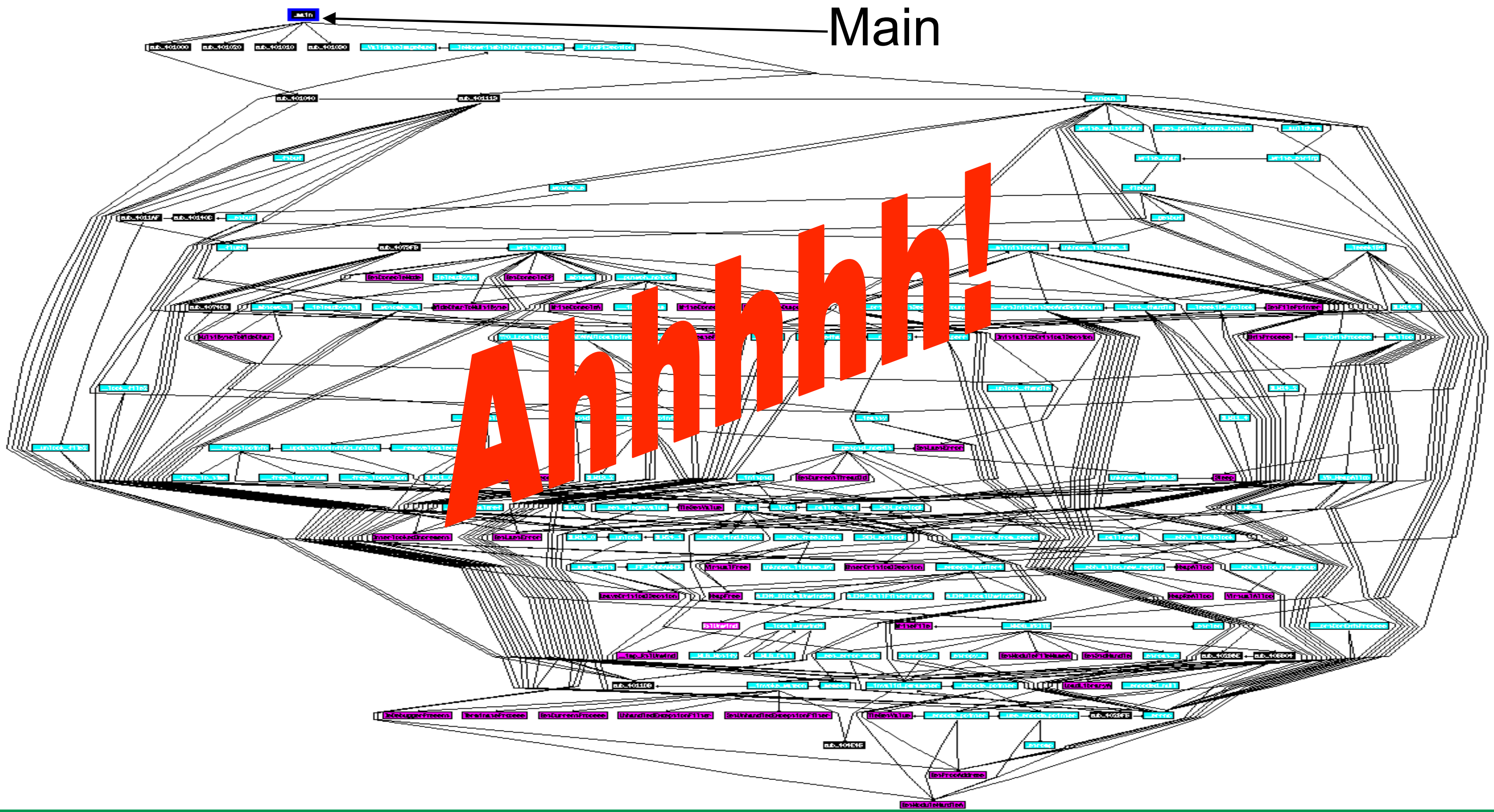
Step 1: The question

- **Audit a piece of Software**
- **Do the developers follow the principles for secure coding?**

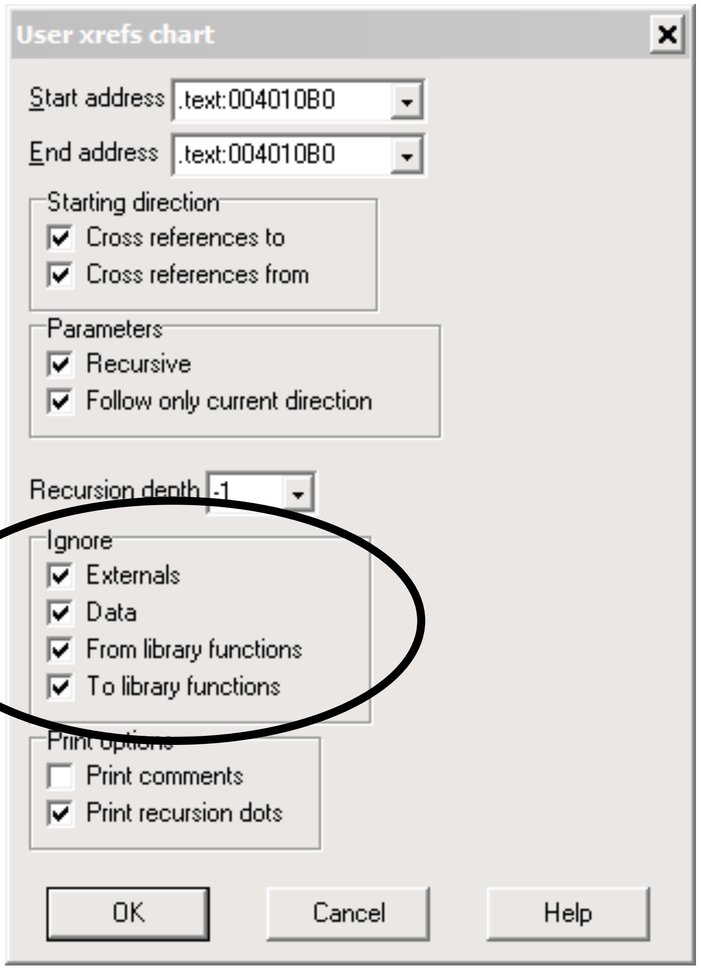
Step 2: Program Flow - Flowchart



Step 2: Program Flow – From main



Step 2: Program Flow – ignore everything but user defined functions


User xrefs chart

Start address .text:004010B0

End address .text:004010B0

Starting direction
- [✓] Cross references to
- [✓] Cross references from

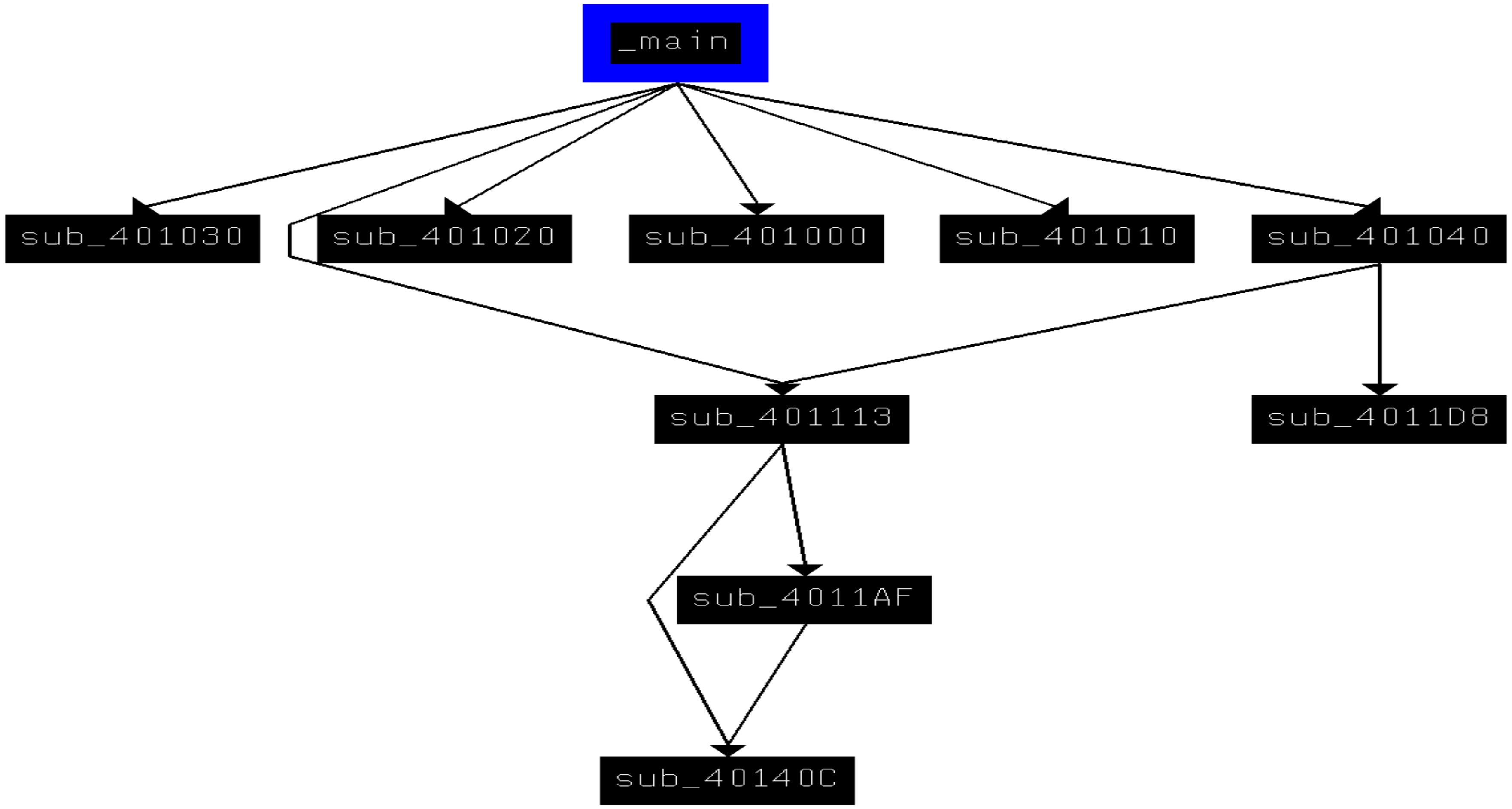
Parameters
- [✓] Recursive
- [✓] Follow only current direction

Recursion depth -1

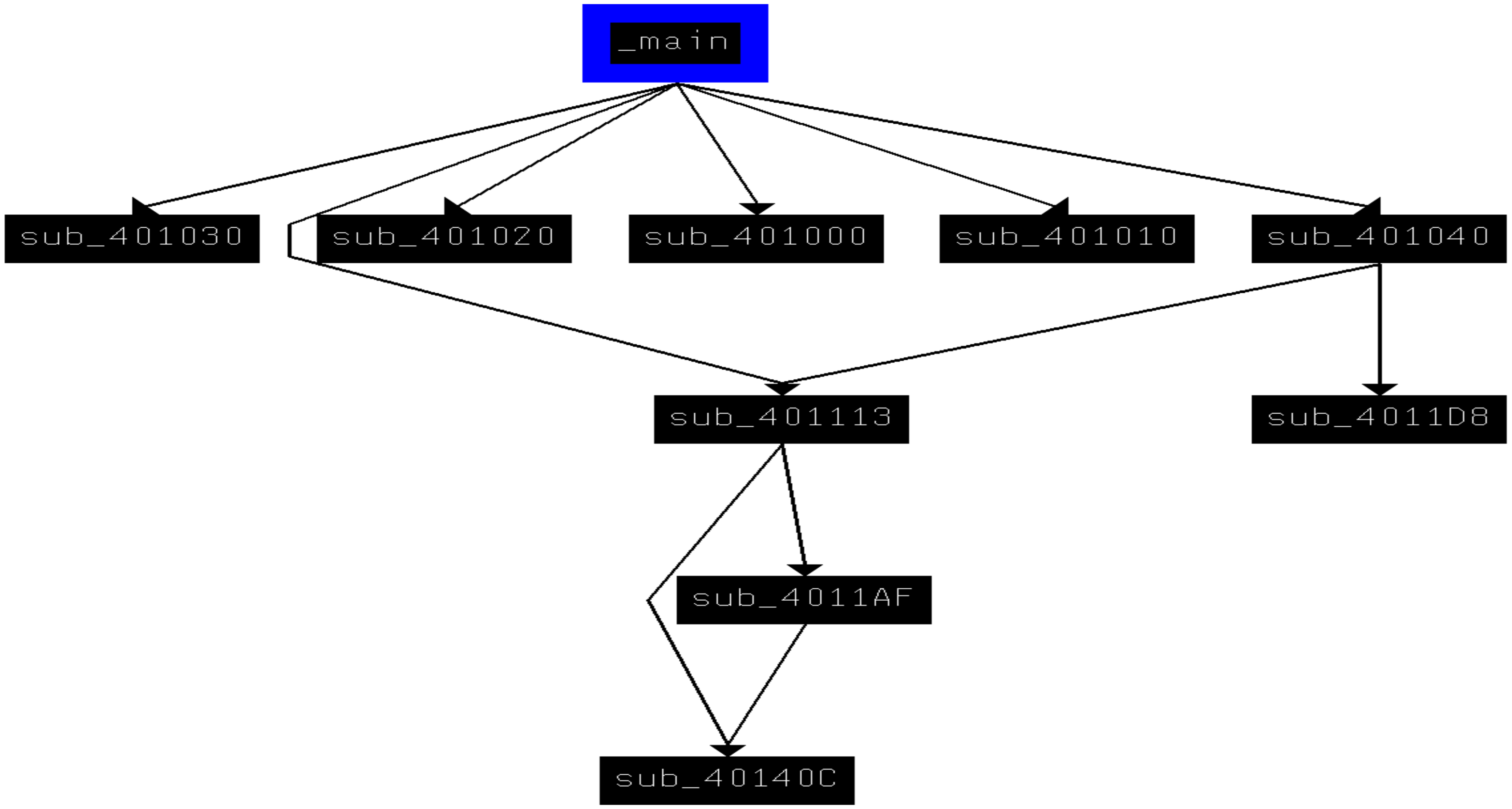
Ignore
- [✓] Externals
- [✓] Data
- [✓] From library functions
- [✓] To library functions

Print options
- [ ] Print comments
- [✓] Print recursion dots

Step 2: Program Flow – Uff ☺



Step 3: Interessting Functions – Here we are ☺



Step 4: Function Prototypes

```
sub_401000 proc near

var_200= byte ptr -200h
arg_0= dword ptr  4
```

Internal variable **Function argument**

Step 4: Function Prototypes

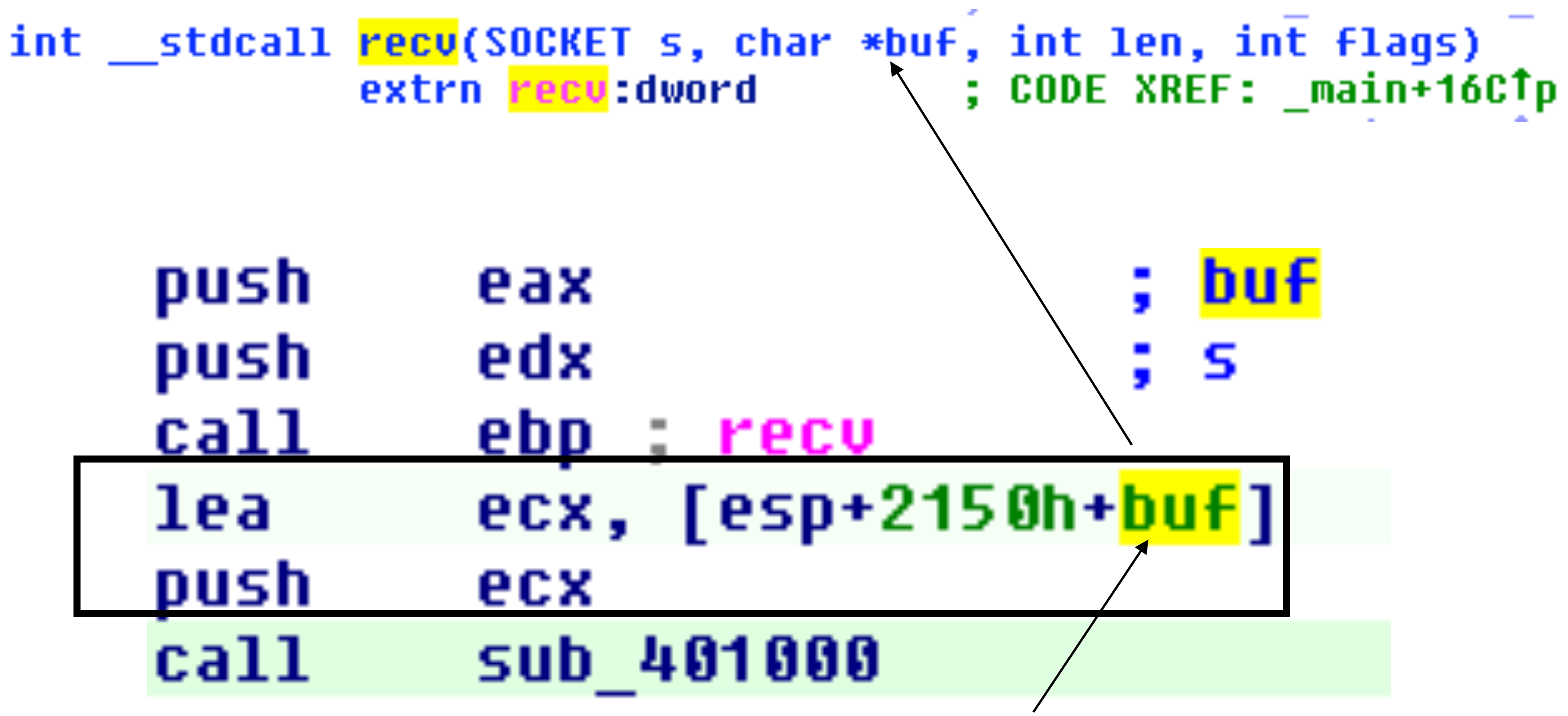
ERNW Living Security.

```
sub_401000 (arg_0);

arg_0= dword ptr  4
```

Which type? Pointer or Integer?

You have to look at the place where the function is called to find out what type is passed to the function

Step 4: Function Prototypes



```
int __stdcall recv(SOCKET s, char *buf, int len, int flags)
              extrn recv:dword           ; CODE XREF: _main+16C↑p
```

```
push     eax                      ; buf
push     edx                      ; s
call     ebp ; recv
lea      ecx, [esp+2150h+buf]
push     ecx
call     sub_401000
```

Yuppieh, it‘s a Pointer to a buffer ☺

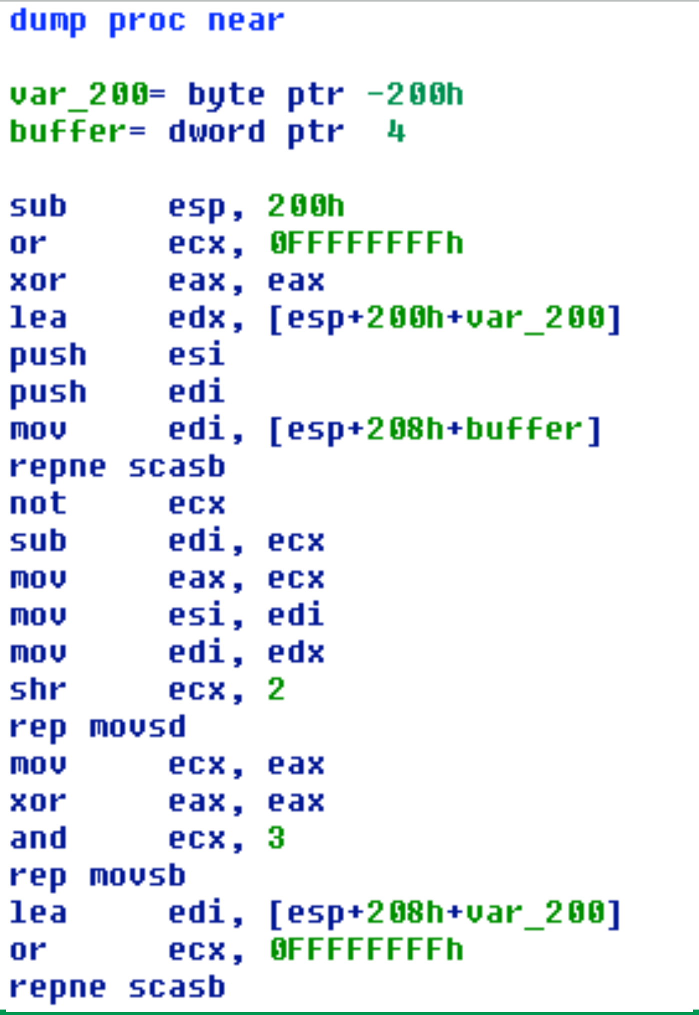
Step 4: Function Prototypes - Here we are

```
sub_401000 (char *arg_0);
```

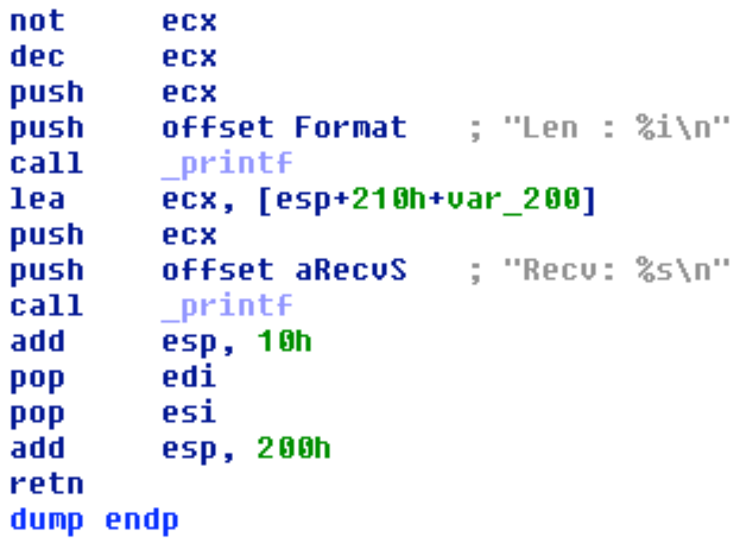
Step 4: Function Prototypes – Or just press F5 and look at the decompiler

```
sub_401000(const char *a1)
{
 char v2;

 ...
}
```

Step 5: Understand what the function is doing - Example 1

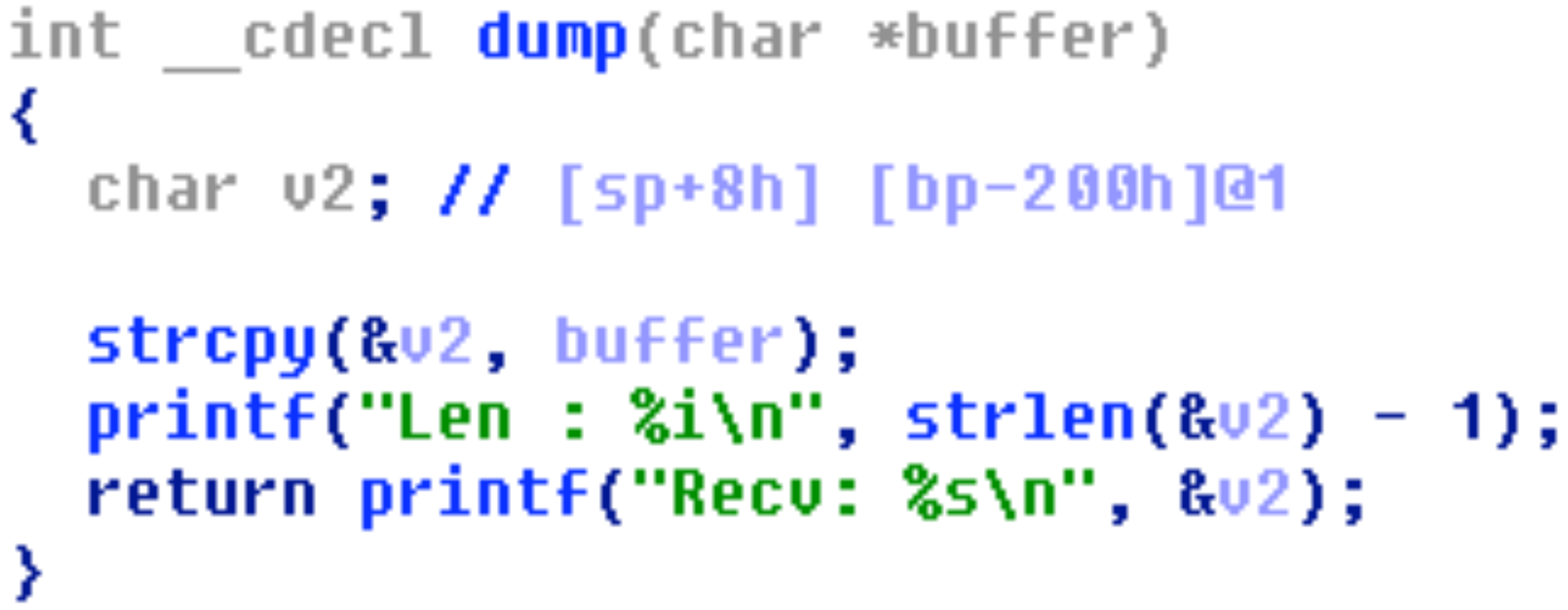


```
dump proc near

var_200= byte ptr -200h
buffer= dword ptr  4

sub     esp, 200h
or      ecx, 0FFFFFFFFh
xor     eax, eax
lea     edx, [esp+200h+var_200]
push    esi
push    edi
mov     edi, [esp+208h+buffer]
repne scasb
not     ecx
sub     edi, ecx
mov     eax, ecx
mov     esi, edi
mov     edi, edx
shr     ecx, 2
rep movsd
mov     ecx, eax
xor     eax, eax
and     ecx, 3
rep movsb
lea     edi, [esp+208h+var_200]
or      ecx, 0FFFFFFFFh
repne scasb
```



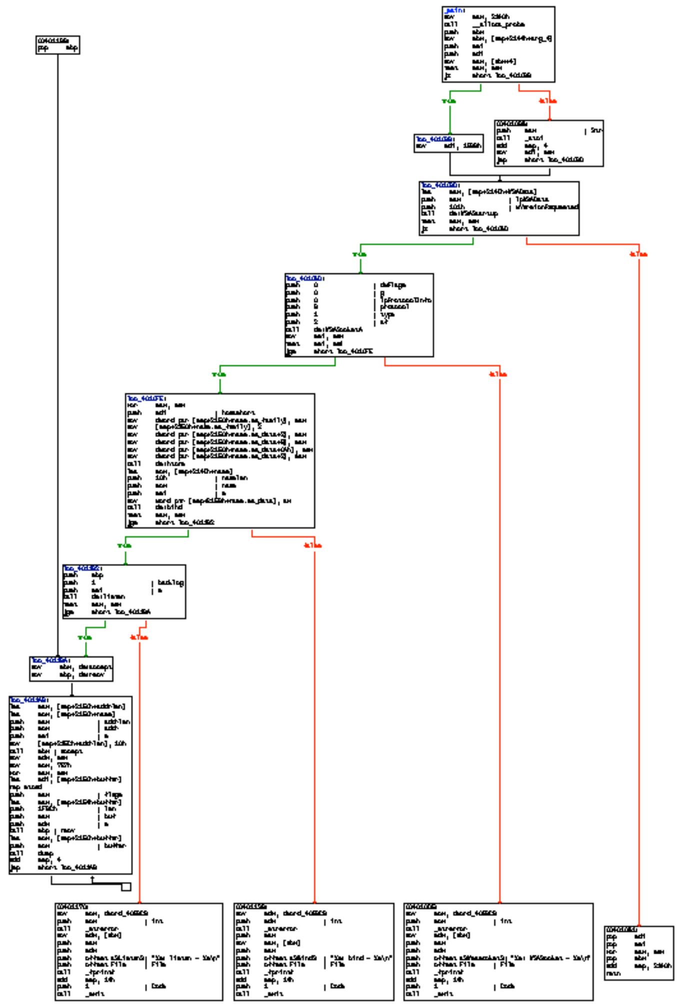
```
not     ecx
dec     ecx
push    ecx
push    offset Format   ; "Len : %i\n"
call    _printf
lea     ecx, [esp+210h+var_200]
push    ecx
push    offset aRecvS   ; "Recv: %s\n"
call    _printf
add     esp, 10h
pop     edi
pop     esi
add     esp, 200h
retn
dump endp
```

Step 5: Understand what the function is doing - Example 1 (Decompiler)



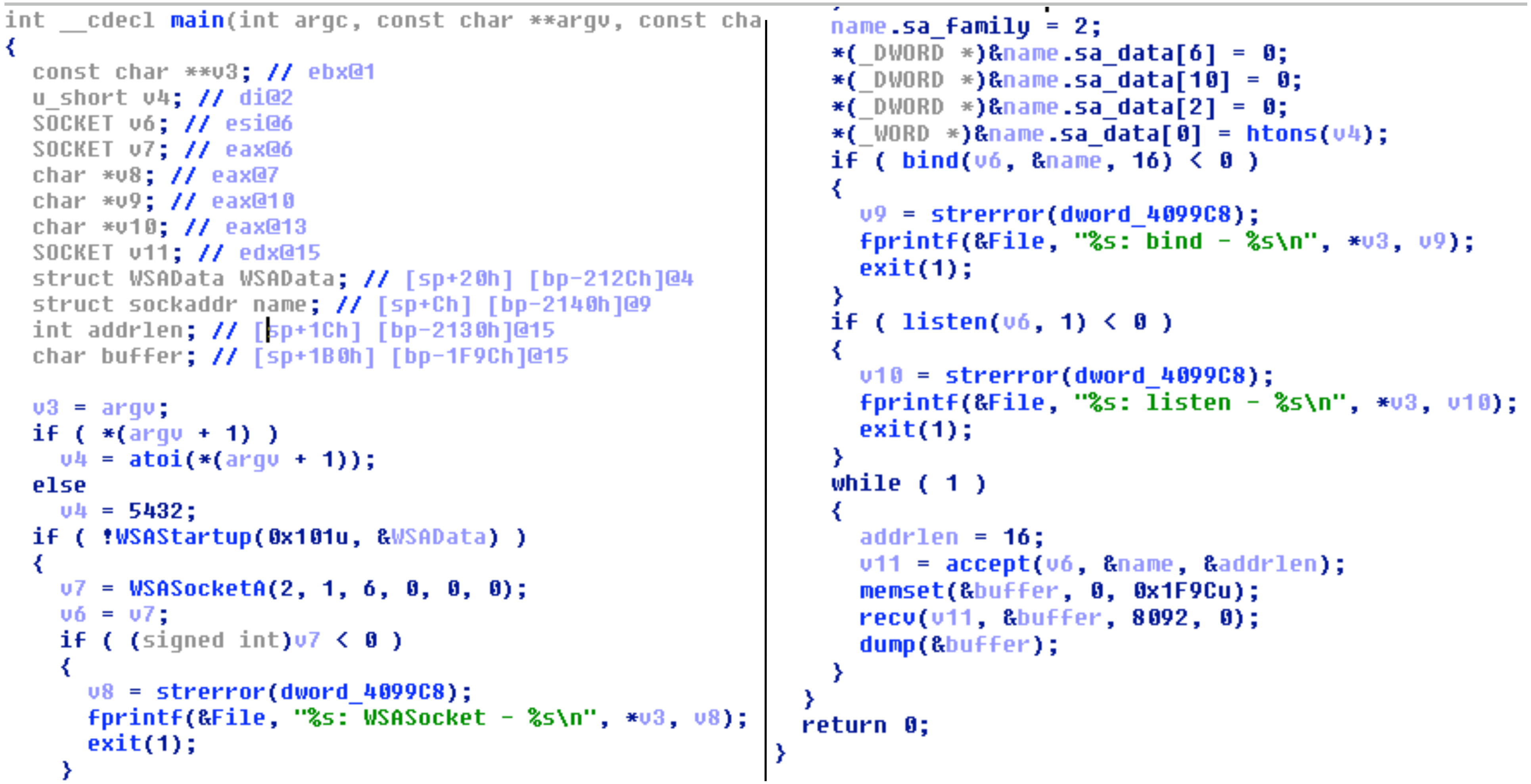
```
int __cdecl dump(char *buffer)
{
  char v2; // [sp+8h] [bp-200h]@1

  strcpy(&v2, buffer);
  printf("Len : %i\n", strlen(&v2) - 1);
  return printf("Recv: %s\n", &v2);
}
```

Step 5: Understand what the function is doing – Example 2



Don’t worry ☺
I can’t read it too
but it’s assembler

Step 5: Understand what the function is doing – Example 2 (Decompiler)



```
int __cdecl main(int argc, const char **argv, const cha
{
  const char **v3; // ebx@1
  u_short v4; // di@2
  SOCKET v6; // esi@6
  SOCKET v7; // eax@6
  char *v8; // eax@7
  char *v9; // eax@10
  char *v10; // eax@13
  SOCKET v11; // edx@15
  struct WSAData WSAData; // [sp+20h] [bp-212Ch]@4
  struct sockaddr name; // [sp+Ch] [bp-2140h]@9
  int addrlen; // [sp+1Ch] [bp-2130h]@15
  char buffer; // [sp+1B0h] [bp-1F9Ch]@15

  v3 = argv;
  if ( *(argv + 1) )
    v4 = atoi(*(argv + 1));
  else
    v4 = 5432;
  if ( !WSAStartup(0x101u, &WSAData) )
  {
    v7 = WSASocketA(2, 1, 6, 0, 0, 0);
    v6 = v7;
    if ( (signed int)v7 < 0 )
    {
      v8 = strerror(dword_4099C8);
      fprintf(&File, "%s: WSASocket - %s\n", *v3, v8);
      exit(1);
    }
```

```
    name.sa_family = 2;
    *(_DWORD *)&name.sa_data[6] = 0;
    *(_DWORD *)&name.sa_data[10] = 0;
    *(_DWORD *)&name.sa_data[2] = 0;
    *(_WORD *)&name.sa_data[0] = htons(v4);
    if ( bind(v6, &name, 16) < 0 )
    {
      v9 = strerror(dword_4099C8);
      fprintf(&File, "%s: bind - %s\n", *v3, v9);
      exit(1);
    }
    if ( listen(v6, 1) < 0 )
    {
      v10 = strerror(dword_4099C8);
      fprintf(&File, "%s: listen - %s\n", *v3, v10);
      exit(1);
    }
    while ( 1 )
    {
      addrlen = 16;
      v11 = accept(v6, &name, &addrlen);
      memset(&buffer, 0, 0x1F9Cu);
      recv(v11, &buffer, 8092, 0);
      dump(&buffer);
    }
  }
  return 0;
}
```

Step 6: Runtime Analysis

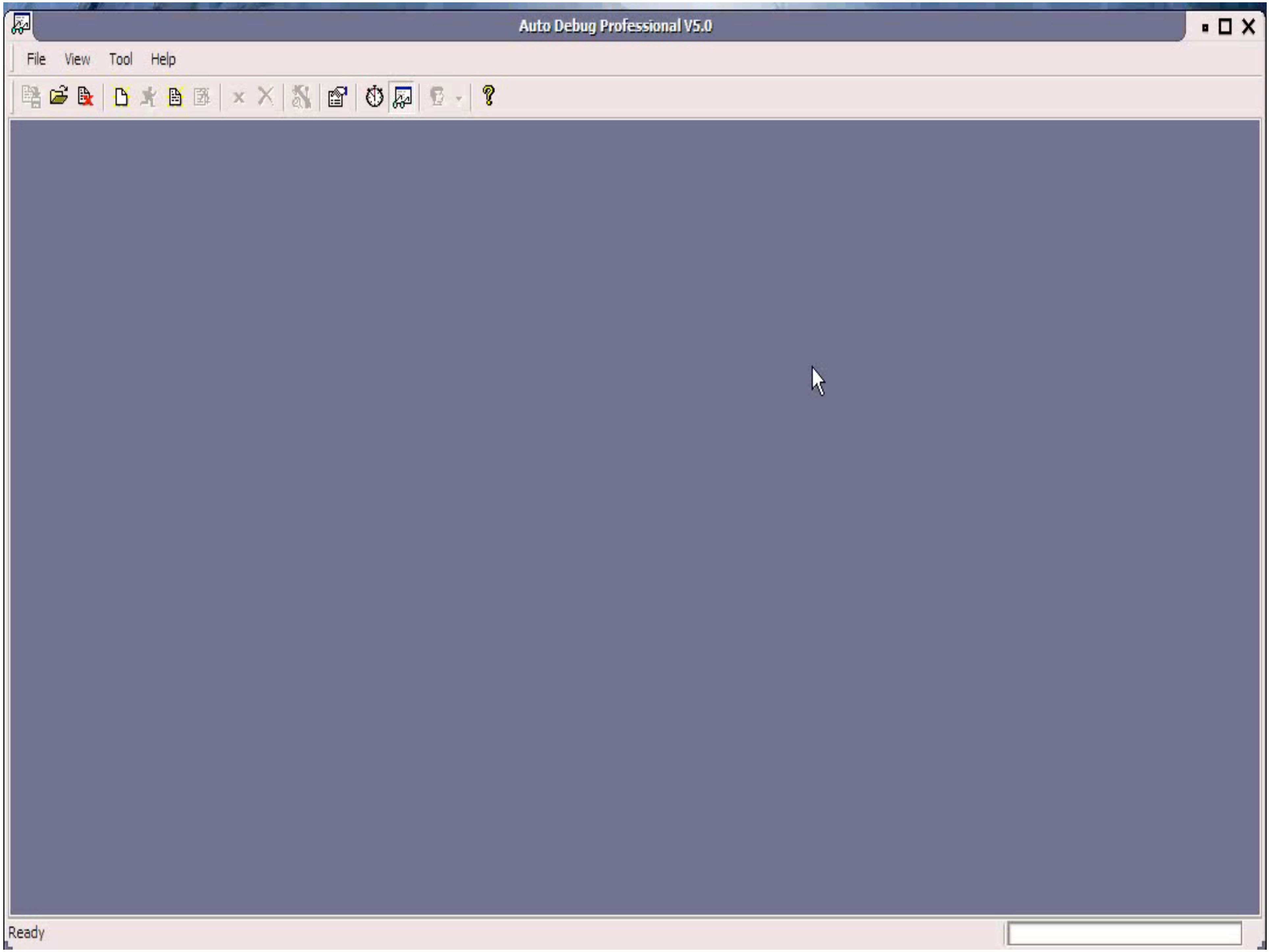
- **Instead of watching Stack variables in a standard debugger, a look at the function call would be much more easier**
- **Autodebug will help to do that, but first autodebug must know the function**

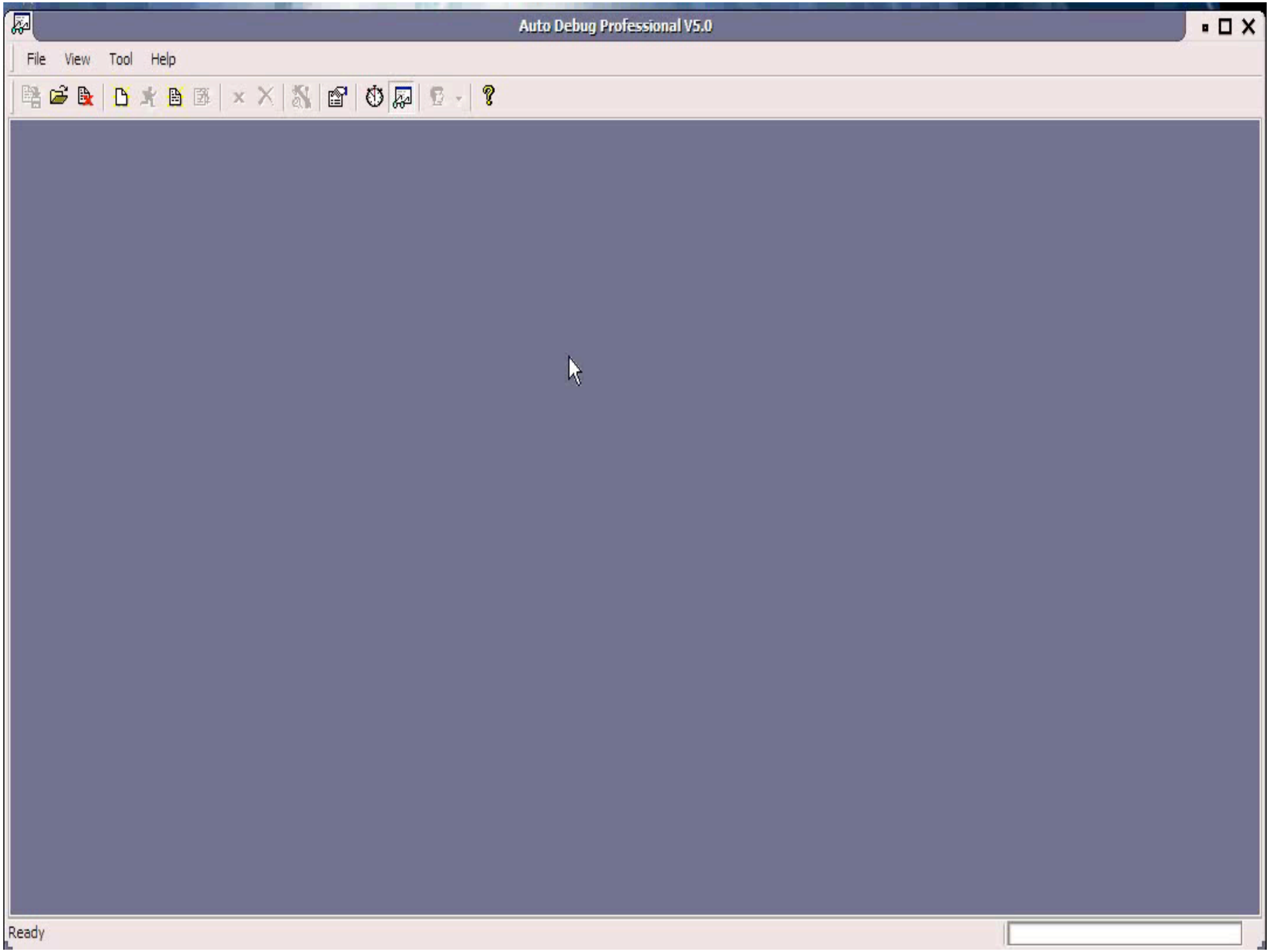
Autodebug without Debug Symbols

- **Step 1: Generate map file within IDAPro**
- **Step 2: Run binary with autodebug**
- **Step 3: Load Map File in autodebug**
- **Step 4: Generate PDB Template (it’s a VS6 Project)**
- **Step 5: Close autodebug**

Autodebug without Debug Symbols

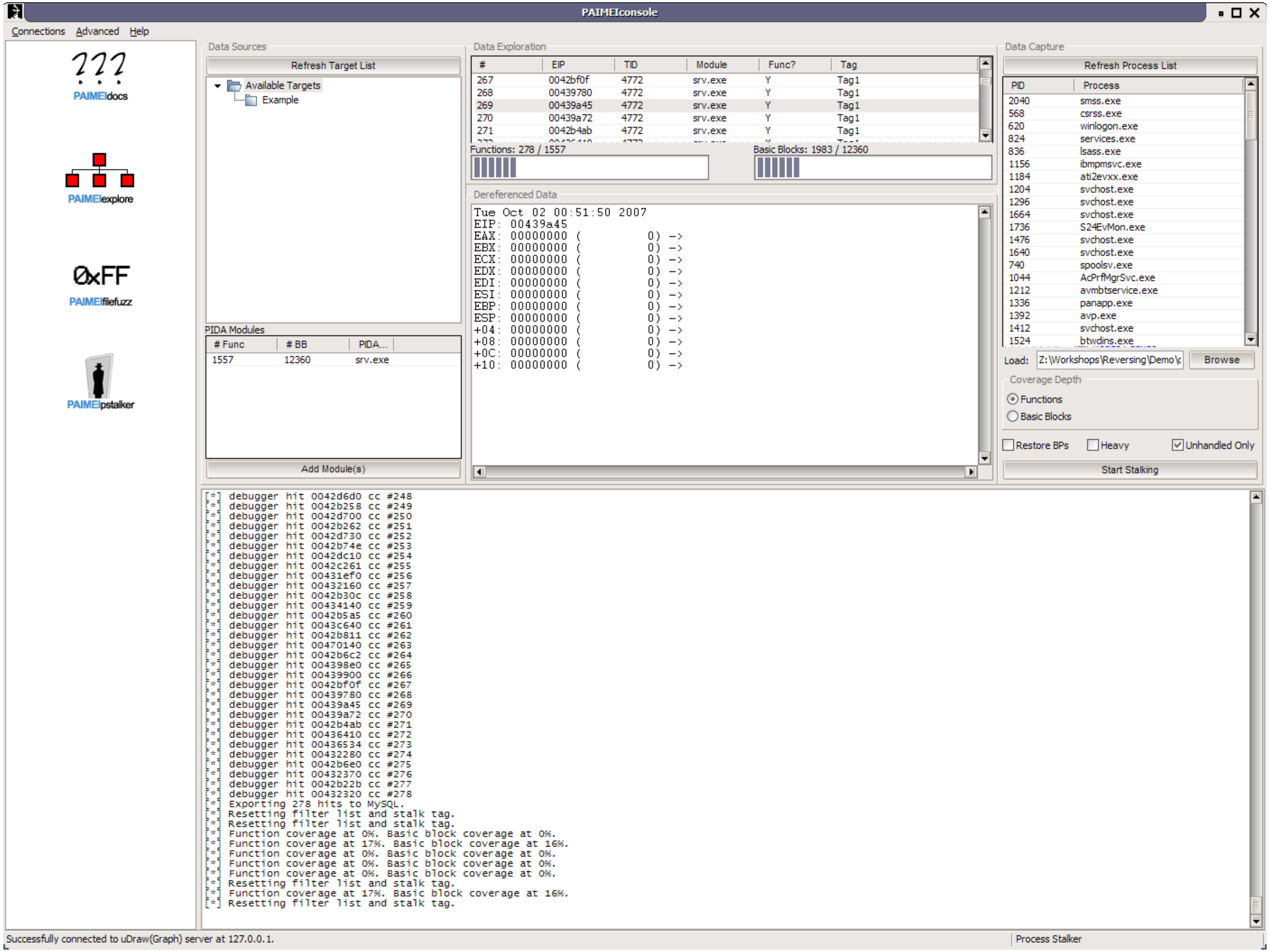
- **Step 6: Fill in the known function prototype (gained from IDAPro / Hex-Rays Analysis) into your PDB template**
- **Step 7: Compile**
- **Step 8: Use PDB File (Program Debug Database) with Autodebug (copy into pdbfiles directory)**
- **Step 9: Load Map File in autodebug**
- **Step 10: Select functions to monitor**
- **Step 11: See which parameters are passed to the function and which values are returned**


Auto Debug Professional V5.0
File
View
Tool
Help
Ready


Auto Debug Professional V5.0
File
View
Tool
Help
Ready

Step 6: Runtime Analysis

- **Code Coverage Analyzer can help to determine which functions are called during runtime**
- **One of PAIMEIs functions is Code Coverage**
- **PAIMEI interacts with IDAPro and has a lot more functionality build in**
- **Code Coverage helps to focus on interessting functions that are called**
- **Find PAIMEI at paimei.openrce.org**



# Func	# BB	PIDA...
1557	12360	srv.exe

#	EIP	TID	Module	Func?	Tag
267	0042bf0f	4772	srv.exe	Y	Tag1
268	00439780	4772	srv.exe	Y	Tag1
269	00439a45	4772	srv.exe	Y	Tag1
270	00439a72	4772	srv.exe	Y	Tag1
271	0042b4ab	4772	srv.exe	Y	Tag1

Functions: 278 / 1557

Basic Blocks: 1983 / 12360

```
Tue Oct 02 00:51:50 2007
EIP: 00439a45
EAX: 00000000 (          0) ->
EBX: 00000000 (          0) ->
ECX: 00000000 (          0) ->
EDX: 00000000 (          0) ->
EDI: 00000000 (          0) ->
ESI: 00000000 (          0) ->
EBP: 00000000 (          0) ->
ESP: 00000000 (          0) ->
+04: 00000000 (          0) ->
+08: 00000000 (          0) ->
+0C: 00000000 (          0) ->
+10: 00000000 (          0) ->
```

PID	Process
2040	smss.exe
568	csrss.exe
620	winlogon.exe
824	services.exe
836	lsass.exe
1156	ibmpmsvc.exe
1184	ati2evxx.exe
1204	svchost.exe
1296	svchost.exe
1664	svchost.exe
1736	S24EvMon.exe
1476	svchost.exe
1640	svchost.exe
740	spoolsv.exe
1044	AcPrfMgrSvc.exe
1212	avmbtservice.exe
1336	panapp.exe
1392	avp.exe
1412	svchost.exe
1524	btwdins.exe

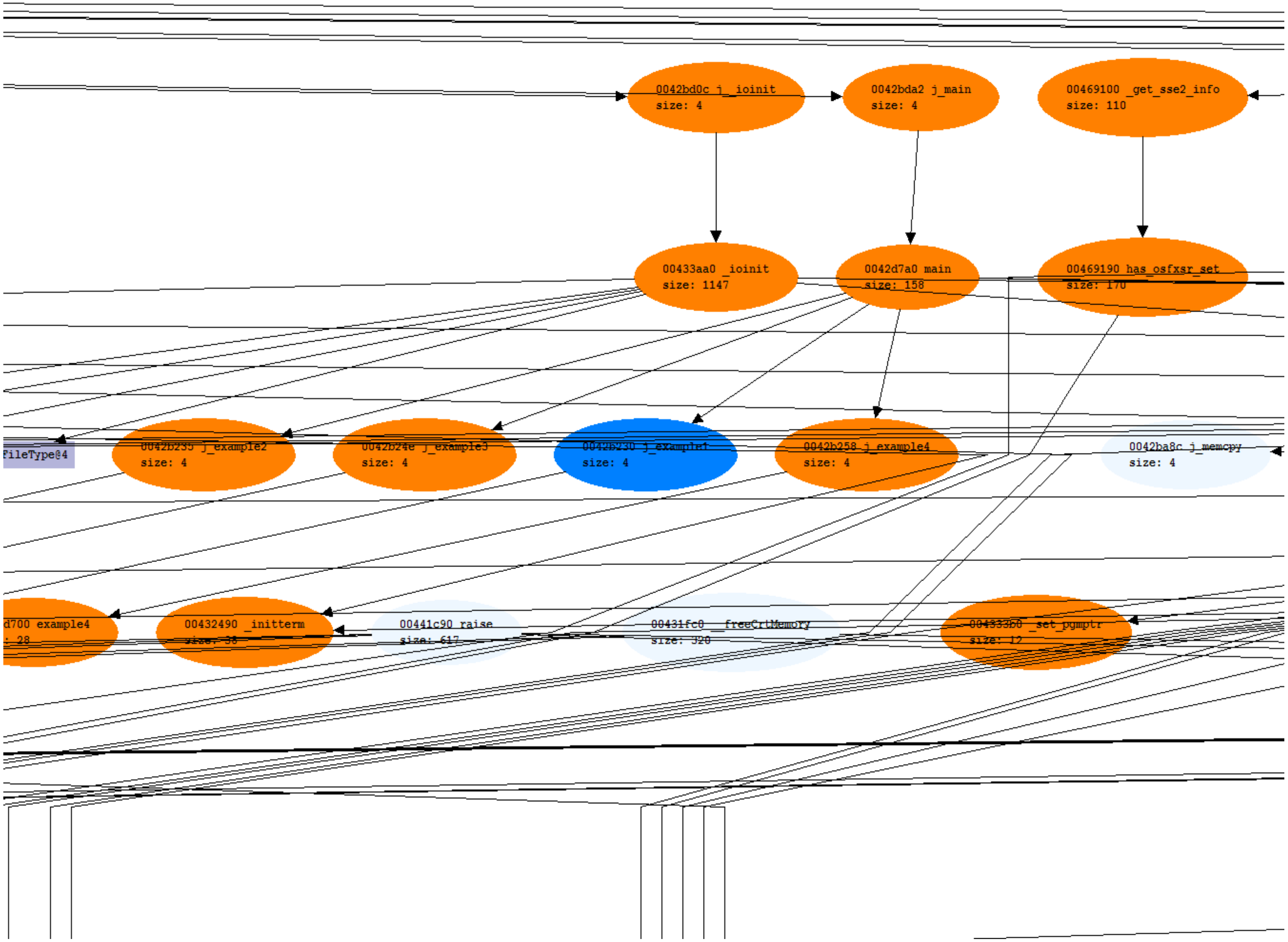
Load: Z:\Workshops\Reversing\Demo\c

Coverage Depth: [✓ Functions] Basic Blocks

Restore BPs ☐ Heavy ☐ Unhandled Only ☑

```
[*] debugger hit 0042d6d0 cc #248
[*] debugger hit 0042b258 cc #249
[*] debugger hit 0042d700 cc #250
[*] debugger hit 0042b262 cc #251
[*] debugger hit 0042d730 cc #252
[*] debugger hit 0042b74e cc #253
[*] debugger hit 0042dc10 cc #254
[*] debugger hit 0042c261 cc #255
[*] debugger hit 00431ef0 cc #256
[*] debugger hit 00432160 cc #257
[*] debugger hit 0042b30c cc #258
[*] debugger hit 00434140 cc #259
[*] debugger hit 0042b5a5 cc #260
[*] debugger hit 0043c640 cc #261
[*] debugger hit 0042b811 cc #262
[*] debugger hit 00470140 cc #263
[*] debugger hit 0042b6c2 cc #264
[*] debugger hit 004398e0 cc #265
[*] debugger hit 00439900 cc #266
[*] debugger hit 0042bf0f cc #267
[*] debugger hit 00439780 cc #268
[*] debugger hit 00439a45 cc #269
[*] debugger hit 00439a72 cc #270
[*] debugger hit 0042b4ab cc #271
[*] debugger hit 00436410 cc #272
[*] debugger hit 00436534 cc #273
[*] debugger hit 00432280 cc #274
[*] debugger hit 0042b6e0 cc #275
[*] debugger hit 00432370 cc #276
[*] debugger hit 0042b22b cc #277
[*] debugger hit 00432320 cc #278
[*] Exporting 278 hits to MySQL.
[*] Resetting filter list and stalk tag.
[*] Resetting filter list and stalk tag.
[*] Function coverage at 0%. Basic block coverage at 0%.
[*] Function coverage at 17%. Basic block coverage at 16%.
[*] Function coverage at 0%. Basic block coverage at 0%.
[*] Function coverage at 0%. Basic block coverage at 0%.
[*] Function coverage at 0%. Basic block coverage at 0%.
[*] Resetting filter list and stalk tag.
[*] Function coverage at 17%. Basic block coverage at 16%.
[*] Resetting filter list and stalk tag.
```

Successfully connected to uDraw(Graph) server at 127.0.0.1.


0042bd0c j__ioinit
size: 4
0042bda2 j_main
size: 4
00469100 _get_sse2_info
size: 110
00433aa0 _ioinit
size: 1147
0042d7a0 main
size: 158
00469190 has_osfxsr_set
size: 170
FileType@4
0042b235 j_example2
size: 4
0042b24e j_example3
size: 4
0042b230 j_example1
size: 4
0042b258 j_example4
size: 4
0042ba8c j_memcpy
size: 4
d700 example4
: 28
00432490 _initterm
size: 38
00441c90 raise
size: 617
00431fc0 __freeCrtMemory
size: 320
004333b0 _set_pgmptr
size: 12

Step 7: Answer the question

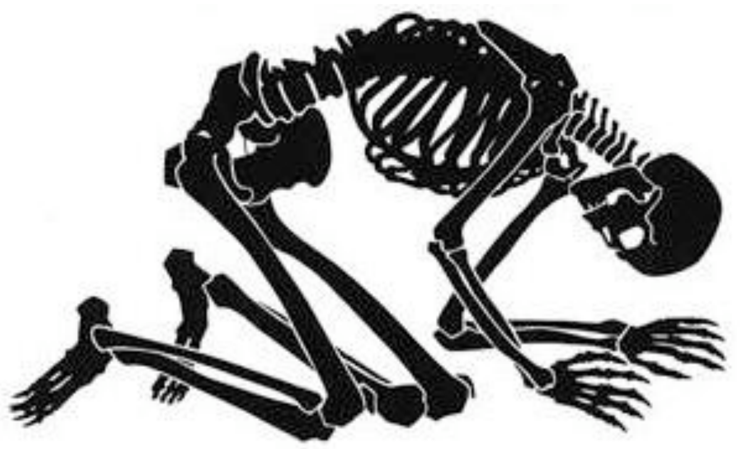
- **Ok, summary anyone?**
- **Is the program secure?**

Final Conclusions

ERNW Living Security.

- This approach works (at least for us ☺)
- Can you answer every question? No you can’t (think of code obfuscation, anti RE functions and so forth where additional steps are needed)
- You don’t have to be an assembler guru to work with this approach, but don’t forget that you still need skilled people
- You still can improve for example code coverage with commercial tools

ERNW Living Security.



Thank’s for your patience

Time left for `questions & answers` ?