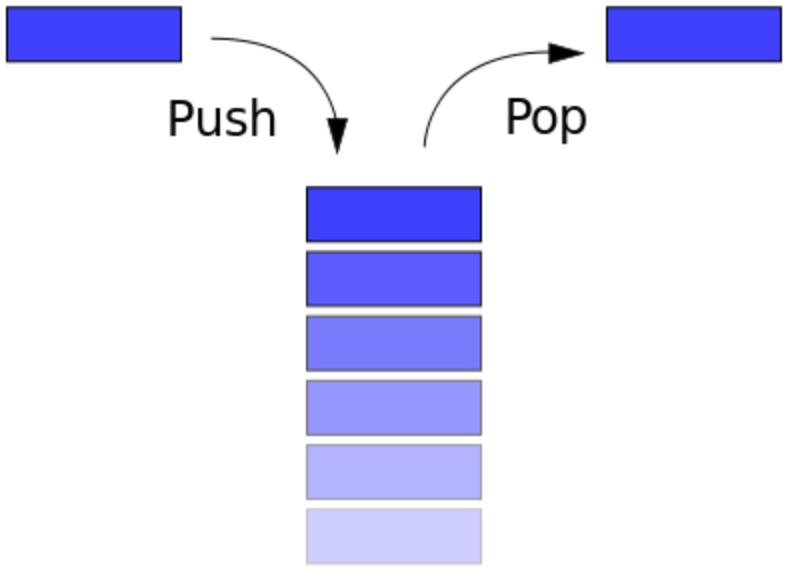
KSG-源码级栈溢出保护工具

李有霖-202021080918

1. Introduction

栈是一种典型的后进先出的数据结构，其操作主要有压栈与出栈两种操作，如下图所示。两种操作都操作栈顶，当然，它也有栈底，程序的栈是从进程地址空间的高地址向低地址增长的。高级语言在运行时都会被转换为汇编程序，在汇编程序运行过程中，充分利用了这一数据结构。每个程序在运行时都有虚拟地址空间，其中某一部分就是该程序对应的栈，用于保存函数调用信息和局部变量。



栈溢出指的是程序向栈中某个变量中写入的字节数超过了这个变量本身所申请的字节数，因而导致与其相邻的栈中的变量的值被改变。这种问题是一种特定的缓冲区溢出漏洞，类似的还有堆溢出，bss 段溢出等溢出方式。栈溢出漏洞轻则可以使程序崩溃，重则可以使攻击者控制程序执行流程。

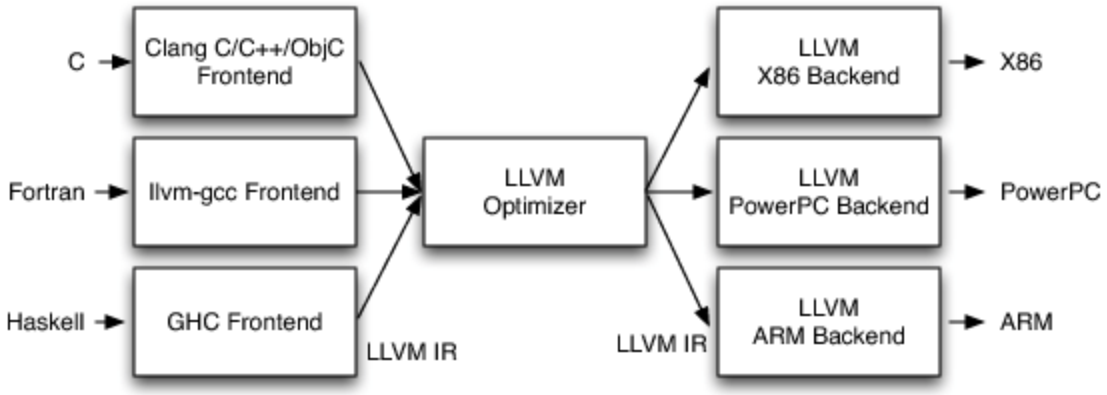
本文借助 LLVM StackProtector 的设计理念，实现了一个源码级的栈溢出保护工具 KSG，主要工作有以下几点：

- 基于 LLVM Pass 实现了一个源码级栈溢出保护工具

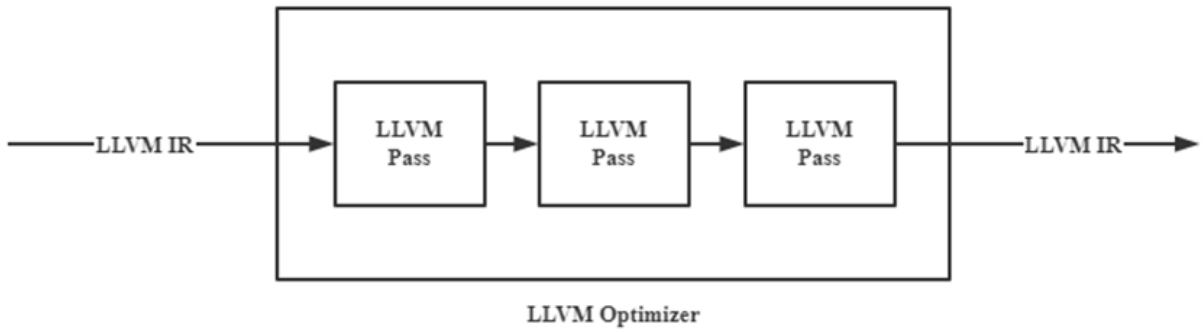
- 使用 inline 插桩、仅保护有数组的函数等策略来减小插桩所带来的开销
- 支持多种后端，能对多种目标架构生成保护代码

2. Background

LLVM 编译器基础设施项目是一组编译器和的工具链技术，其可用于开发前端任何编程语言和一个后端为任何指令集架构。LLVM围绕一种独立于语言的中间表示（IR）设计，该中间表示用作一种便携式的高级汇编语言，可以通过多次遍历其中的各种转换来对其进行优化。LLVM 的三段式设计框架如下图所示。KSG 的核心代码处于中间层，中间层 IR 经过前端的分析，拥有相当丰富的语义信息可供进一步分析，并且由于和前端后端解耦，可以轻松适配多种高级语言和目标架构。

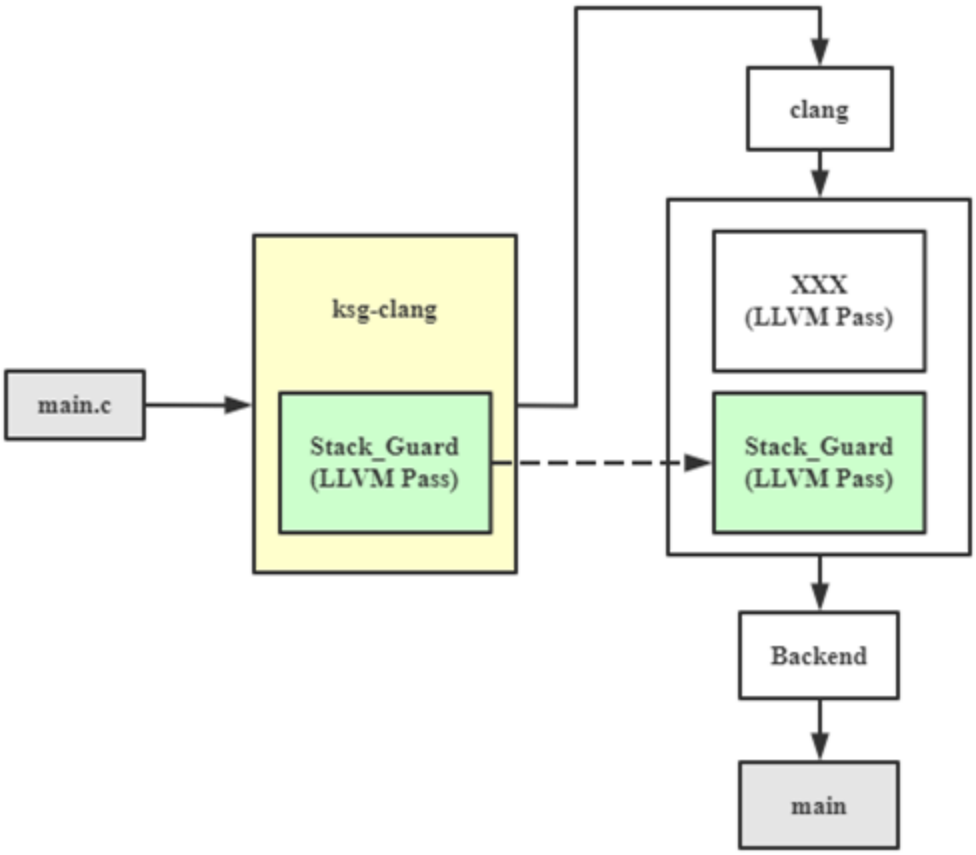


中间层的具体设计实现可以简化为下图，可以看作是由多个 LLVM Pass 组成，每个 Pass 代表一种优化方案，其遍历所有的 IR 并在其上进行操作。Pass 执行构成编译器的转换和优化，它们构建这些转换使用的分析结果，是编译器代码的结构化技术。KSG 的主要功能也是由一个 LLVM Pass 提供。

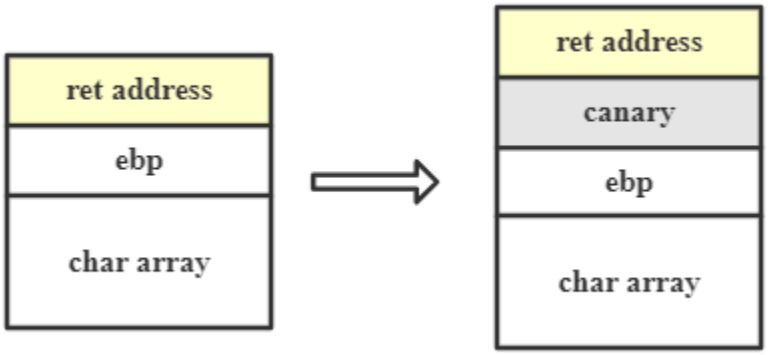


3. Design

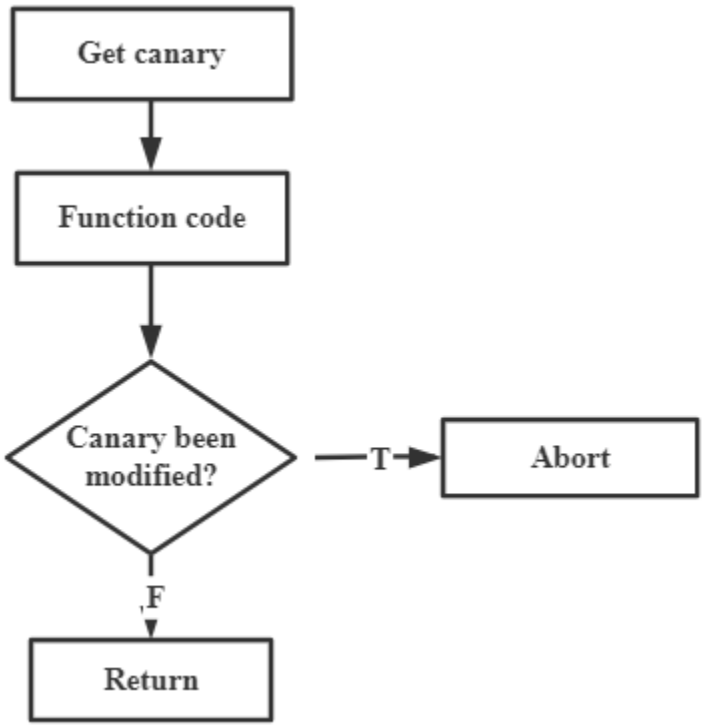
KSG 的框架如下图所示，核心代码实现了一个 LLVM Pass，其外部实现了一个包装器对 clang 进行封装，使得用户可以像使用 clang 一样使用 KSG。KSG 会在编译的优化阶段插入编写好的 Pass，由 Pass 在 IR 层对被保护程序进行插桩，最后交由后端生成可执行程序。这一部分的设计参照了项目 AFLPlusPlus。



沿用已久的栈溢出保护是一个很简单的技术，我们也没有必要设计新的方法。正如前面提到的本文借助 LLVM StackProtector 的设计理念，所以在方法和实现上都会尽可能还原 LLVM StackProtector。对于栈帧的改变如图所示，在函数的局部变量和返回地址之间放置了一个名为 canary 的全局变量，该变量在程序初始化时初始化为一个随机数值。



函数执行流程改变如下图所示，每次函数返回的时候检查栈上 canary 是否和全局变量一致，不一致则判断为 canary 被修改，发生了栈溢出。这样的设计简单易于实现，不会给程序增加太多运行时的开销，但也有和 LLVM Protector 一样的缺点：不能检查是否存在栈溢出覆盖了局部变量。



4. Implementation

LLVM Pass 提供了接口 runOnFunction，其在遍历 IR 的每个函数时被调用。首先判断函数中是否包含局部变量数组的分配，如果函数中存在局部变量数组则对函数进行保护。

保护的第一步是在函数头部插入一个局部变量的分配指令，然后插入一个赋值指令将全局 canary 赋值给局部 canary。

```
IRBuilder<> builder_entry(&F.getEntryBlock().front());
AllocaInst * canary_slot = builder_entry.CreateAlloca(Type::getInt32Ty(context));
Value * t1 = builder_entry.CreateLoad(Type::getInt32Ty(context), gcanary, true);
builder_entry.CreateStore(t1, canary_slot);
```

需要注意一个函数可能包含几个 return 指令，所以需要遍历所有的基本块，在末尾是 return 指令的基本块上加入检查 canary 的操作。检查是 inline 方式实现的，相对较为复杂。首先将基本块按照 return 指令分割为两个基本块，在前一个基本块后插入比较 canary 的语句，将两个基本块的无条件连接跳转删除，改为根据判断结果的条件跳转，并将判断失败的分支连接到检查失败基本块。

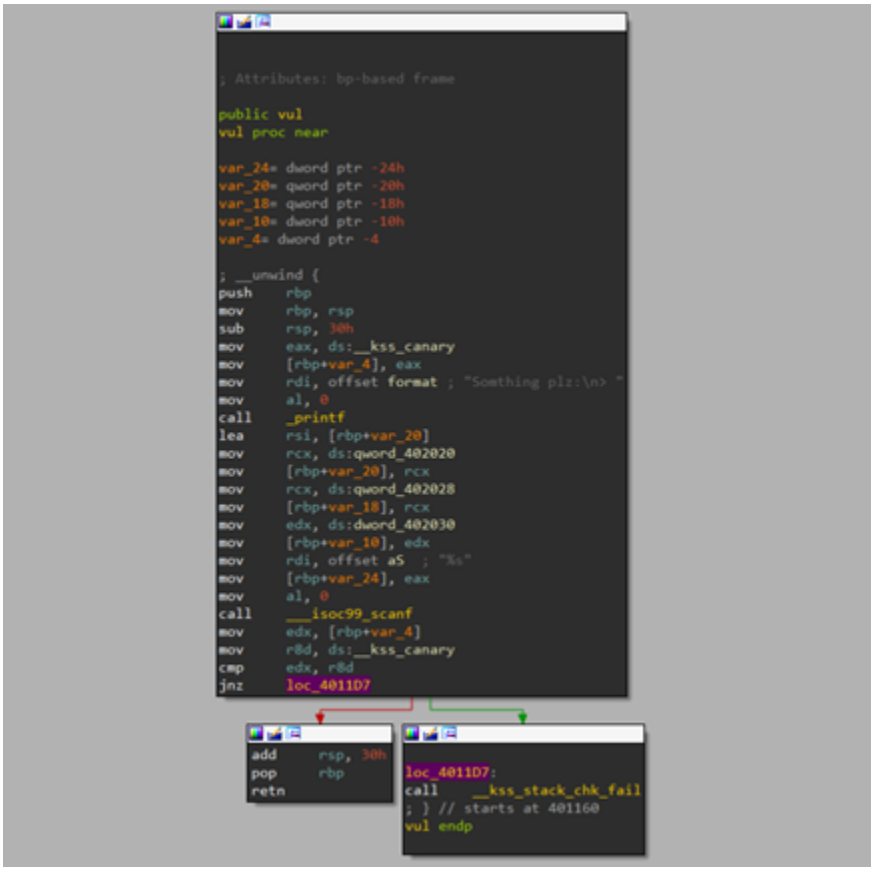
```
for (Function::iterator I = F.begin(), E = F.end(); I != E;)
{
    BasicBlock &BB = *I++;
    ReturnInst *IST = dyn_cast<ReturnInst>(BB.getTerminator());
    if (IST)
    {
        BasicBlock *FailBB = getFailBB(F);

        BasicBlock *NewBB = BB.splitBasicBlock(IST->getIterator());
        BB.getTerminator()->eraseFromParent();
        NewBB->moveAfter(&BB);

        IRBuilder<> builder_epilogue(&BB);
        Value *canary_local = \
            builder_epilogue.CreateLoad(Type::getInt32Ty(context), canary_slot, true);
        Value *canary_global = \
            builder_epilogue.CreateLoad(Type::getInt32Ty(context), gcanary, true);
        Value *Cmp = builder_epilogue.CreateICmpEQ(canary_local, canary_global);
        builder_epilogue.CreateCondBr(Cmp, NewBB, FailBB);
    }
}
```

5. Experience

被保护程序代码如下图所示，main 函数负责循环调用一个有栈溢出漏洞的函数 vul，在 vul 中申请了局部变量数组，并且不安全地调用了 scanf 函数。

```c
#include "stio.h"
void vul(){
    printf("Something plz:\n> ");
    char tmp[8] = "pwn";
    scanf("%s", tmp);
}
int main(){
    while(1){
        vul();
    }
    return 0;
}
```

插桩后的 vul 控制流图如图所示，可以看到函数开头获得 canary 存入栈上，并在函数结尾有一个很明显的比较分支操作。其比较局部 canary 和全局 canary 的数值，并在检查到 canary 被修改时调用 chk_fail 函数终止程序。



上述汇编代码的 C 语言表示如下图所示，可以更直观的看到程序逻辑，并验证逻辑的正确性。

```
__int64 vul()
{
  __int64 result; // rax
  __int64 v1[2]; // [rsp+10h] [rbp-20h] BYREF
  int v2; // [rsp+20h] [rbp-10h]
  int v3; // [rsp+2Ch] [rbp-4h]

  v3 = _kss_canary;
  v1[0] = 0x6E7770LL;
  v1[1] = 0LL;
  v2 = 0;
  printf("Somthing plz:\n> ");
  result = __isoc99_scanf("%s", v1);
  if ( v3 != _kss_canary )
    _kss_stack_chk_fail();
  return result;
}
```

下图显示了 KSG 对漏洞程序的保护，首先使用 clang 编译程序生成有漏洞的二进制程序，然后我们给程序一个超长的输入，由于发生栈溢出覆盖到了返回地址，所以程序段错误崩溃了。随后使用 ksg-clang 对编译程序，再次给程序违法的输入时，可以发现程序已经检查到了栈溢出漏洞，并在返回到上级函数之前输出提示信息并主动终止了程序，阻止了进一步的漏洞利用。

```
❯ clang main.c -o main
❯ ./main
Somthing plz:
> oooops_this_tring_is_too_long
[1]    897694 segmentation fault (core dumped)  ./main
❯ ksg-clang main.c -o main
[+] Implemente @ vul.
❯ ./main
Somthing plz:
> oooops_this_tring_is_too_long
[!] Stack overflow detected!
[1]    898001 abort (core dumped)  ./main
```

最后我测试了 KSG 对多架构的适配情况，只需要稍微更改编译参数就可以添加对 MIPS 架构的支持，生成的程序如下图所示，可以看到漏洞程序也被很好的保护了。

```
addiu   $sp, -0x30
sw      $ra, 0x28+var_s4($sp)
sw      $fp, 0x28+var_s0($sp)
move    $fp, $sp
lw      $v0, _fbss
sw      $v0, 0x28+var_4($fp)
lui     $v0, 0x40  # '@'
addiu   $a0, $v0, (aSomthingPlz - 0x400000)  # "Somthing plz:\n> "
sw      $at, 0x28+var_10($fp)
jal     printf
nop
lui     $at, 0x40  # '@'
addiu   $v1, $at, (byte_400B2C - 0x400000)
ulw     $a0, (dword_400B30 - 0x400B2C)($v1)
sw      $a0, 0x28+var_8($fp)
lwl     $a0, byte_400B2C
lwr     $a0, (byte_400B2F - 0x400B2C)($v1)
sw      $a0, 0x28+var_C($fp)
lui     $at, 0x40  # '@'
addiu   $a0, $at, (aS - 0x400000)  # "%s"
addiu   $a1, $fp, 0x28+var_C
sw      $v0, 0x28+var_14($fp)
jal     __isoc99_scanf
nop
lw      $at, 0x28+var_4($fp)
lw      $v1, 0x28+var_10($fp)
lw      $a0, 0x1080($v1)
bne     $at, $a0, __kss_stack_chk_fail
nop
j       loc_400854
nop
```

进一步在 qemu 中运行这个程序可以发现 KSG 也很好的发挥了作用。

```
> qemu-mips-static ./main
Somthing plz:
> ooops_too_long_string_hacked_by_k1ll3r
[!] Stack overflow detected!
qemu: uncaught target signal 6 (Aborted) - core dumped
[1]    908926 abort (core dumped)  qemu-mips-static ./main
```

6. References

- https://github.com/RLee063/Courses/tree/master/k-s-g
- http://www.aosabook.org/en/llvm.html
- https://llvm.org/docs/WritingAnLLVMPass.html
- https://github.com/AFLplusplus/AFLplusplus