CMPT 479/980

Defense: Detecting and repairing control-flow hijacking attacks

Xiyu Zhang

Luoxi Meng

2020
Spring

Abstract

Control-hijacking is considered as one of the most dangerous attacks. DIRA provides a comprehensive way of control-hijacking attack detection, identification, and repair. Our aim is to partial reproduce DIRA. We use both dynamic and static techniques in this project to detect overwrites to return address and function pointers.

Content

Introduction

A control-hijacking attack overwrites some data structures in a victim program that affects its control flow, and eventually hijacks the control of the program and possibly the underlying system. Once an attacker takes control of the victim program, he can then invoke any operations with privilege of the exploited process.

One of the most used attack techniques of control-hijacking attacks is buffer overruns. A buffer overruns occurs when a program attempts to read or write beyond the end of a bounded array (a buffer). Unlike Pascal, Ada, Java and C#, who will check for buffer overruns, in the runtime environment of C and C++, no such checking is performed. Attackers can exploit this defect by using the buffer overruns to overwrite the data structures that affect the control flow. Common attacks overwrite the return address, function pointers and base pointers.

Over the decades, there are a great number of researches done dedicated to control-hijacking attack. Some focus on detecting potential buffer overruns based on static analysis techniques. Others use program transformation techniques to convert applications into a form that can either detect control-hijacking attack or prevent control-sensitive data structure from being modified at run time.

Regardless of their approaches, most of them can only determine whether a program is (potentially) under control-hijacking attack, but could not actively repair the program once it has been attacked. DIRA, however, covers both detection and recovery of a program against control-hijacking attacks (as we can see in the next section).

Our project aims at partially reproducing the defence strategy used by DIRA. Our strategy includes 2 parts. First, automatically detecting and repairing control-hijacking attacks that overwrites return addresses or base pointers. Second, statically generating an AST tree for the source code, modifying the AST tree and recompiling it.

Background and Related Work

DIRA consists of three parts as mentioned before, namely Detection, Identification and Recovery, therefore in this section, we will mention 2 related programs for each part.

- Detection
    - StackGuard: it places a word “canary” right before the return address. When function returns, check if return address is modified.
    - RAD: it copies the return address to a buffer called the return address repository which is protected both sides by applying *myprotect()* system call.
- Identification
    - Buffercup: it prevents malicious codes from entering the system by identifying signatures of which.
    - Horgan:it analyzesthe execution trace of a compromised programs.
- Recovery
    - Igor: it saves modified memory pages at each periodic state checkpoint.
    - Spyder: it records the program counter and the old values of all variables that the current instruction will change.

Many static tools have been developed to detect buffer overruns. Important tools include Prefast, Prefix, ESPx, BOON, Splint, ARCHER, and PolySpace. Most of these detectors apply an exhaustive search over the program for potential violations of buffer access, and maybe require annotations to achieve precision.

Problem Statement

- DIRA stands for detection, identification and repair. For this project we will omit the identification part.
- We will use memory logging to track the control-sensitive data structures, including return address, base pointer and function pointers.
- What our program should do:
    - Static Part
        - parse the source codes based on static analysis techniques, generate AST trees and modify the AST tree to prevent potential buffer overruns.
        - convert the modified AST tree back to new source codes.
        - recompile the modified source code to prevent buffer overruns.
    - Dynamic Part
        - at compile time, insert assembly codes to create data structure for buffering control-sensitive data structures (return address, base pointers).
        - at runtime, buffer the return addresses and base pointers in the function prologues and in the function epilogue, compare the value with the buffered value. If there is a mismatch, restore the data structure to recover it.

Methodology

Candidate Tools & Languages

We researched on a variety of different tools we could make use of for our implementation . The following are the tools and languages we considered that could be useful for us.

Tools

- GNU GCC-Plugin
  GCC-Plugin is available for GCC version 4.5.0 or higher. It is a tool that comes with GCC. We thought since GCC-Plugin is also developed by GNU and it is specifically designed for GCC, it would make our development much more easier.
- LLVM
  LLVM stands for Low Level Virtual Machine, it is a compiler infrastructure project which provides many powerful tools.
- GCC-Python -Plugin
  Python also has a set of APIs for GCC-Plugin development. Developing in Python is relatively easier than other languages, we therefore considered it as one of our options.

Languages

- C
  Since GNU GCC-Plugin is written in C, we considered C as our major language.
- Python
  If we were to adopt the GCC-Python-Plugin approach, we would for sure need to write Python scripts.
- x86-64/x64 Assembly
  Since our development will change the machine code that GCC generates, we thought it would be possible that we need to write some assemblies.

Selected Tools & Languages

Tools

- GNU GCC-Plugin

Our final product used GNU GCC-Plugin as our main development tool. However, our initial attempt was conducted using LLVM. The reason why we gave up on it was primarily due to its large code base. We are all very new to LLVM, even though LLVM provides numerous functionalities, it is extremely difficult to find what we require in the sea of all tools present. For this reason, we found ourselves spending too much time reading documentations and trying out examples but making actual progress towards our project. On the other hand, GNU GCC-Plugin is much more specific for our goal. We therefore picked GNU GCC-Plugin over the other tools we initially considered.

Languages

- C

  We used C as our programming language. We used a library "RTL" in the GNU GCC-Plugin dev package. This library provides a way for us to locate function prologue and epilogue. It also allows us to use C code to alter the final machine code generated by GCC. The details will be discussed in the next section.

- x86-64

  x86-64 was not directly used in our case, yet we still had to write scripts so that we know how to write "rtl".

- RTL(Register Transfer Language)

  The last part of the compiler work is done on a low-level intermediate representation called Register Transfer Language (RTL). In this language, the instructions to be output are described, one by one, in an algebraic form that describes what the instruction does. In out project, we create RTL expressions (RTX) to represent the assembly codes we want to insert in the final assembly code.

- Python

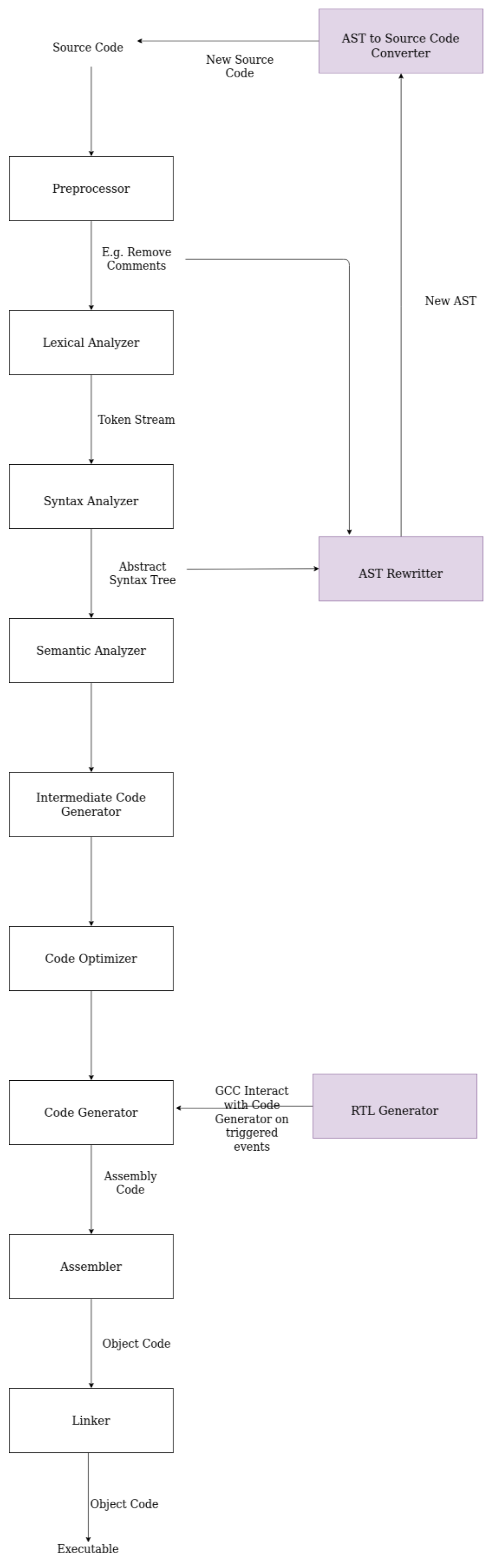
  We used pycparser for C99 to conduct static analysis. These analysis include generating AST tree from source files; modifying AST tree and recompiling the tree back into C99 code

Other Important Tools

- **objdump, gdb, ped**

  These tools help to check how the RTL representation affects the resulting assembly code. When an error (usually internal compiler error) occurs, we emit the RTL instructions one by one, and check the assembly of the compiled file to see what happens. GDB and peda helps a low to track the value and state of registers and contents in stack.

Implementation Details



For this section, we will separate our implementation into two parts. Roughly speaking, one is on lower level (dynamic) and the other is on higher (static). For better understanding, please see Figure 1 we prepared on the left.

Figure 1 shows the phases gone through for source files to be compiled to executable files. We will not reiterate what each step does.

The purple blocks show what our program does. The first part of the program will take an output from C preprocesser and the corresponding AST, then the program does some modification to the source code, then it converts the modified AST back to new source code which is fed into the compiling stack again, then there comes the second part of our program. This first part of the program takes care of function pointer overflow.

The second part of the program happens at the last part of the compiler's work. When entering this part, the source codes have been through a great number of 'passes' (processes that transfer codes from higher level representation to lower level representation), and now are in RTL. Our functions are triggered by events such as function call prologue and function call epilogue, on such events occuring, our program emits RTL instruction into the RTL representation before or after the triggered point. This will generate a new RTL representation, and then the Assembler will assemble based on our new RTL representations. This part of our program takes care of return address overflow and recovery.

Note that our current program design (depicted in Figure 1) differentiates from our initial plan. Our initial plan was to use RTL for both function pointer overflow and return address overflow, however we were not able to locate function pointer declaration, assignment or usage using RTL. It is likely that RTL does not take notes for such events. We will cover the details in the following subsections.

We will now describe in more details of our design, we will first describe how we used the tool we selected and then come to the description of the two parts of our program separately.

GCC-Plugin

GCC-Plugin is the tool we used to integrate the two parts of our program together. The first step is to initialize the plugin. In the initialization step, we can register callback functions when certain events occur. There are many predefined events on different levels during the compiling in GCC-Plugin. To give a few examples:

- PLUGIN_FINISH_DECL: this event is triggered upon the completion of variables declaration. This is an event after a syntax analyzer.
- PLUGIN_FINISH: this event is triggered right before GCC exits.

Because callback functions are user-defined, we can then change the behavior of GCC by invoking those callback functions when our desired events occur.

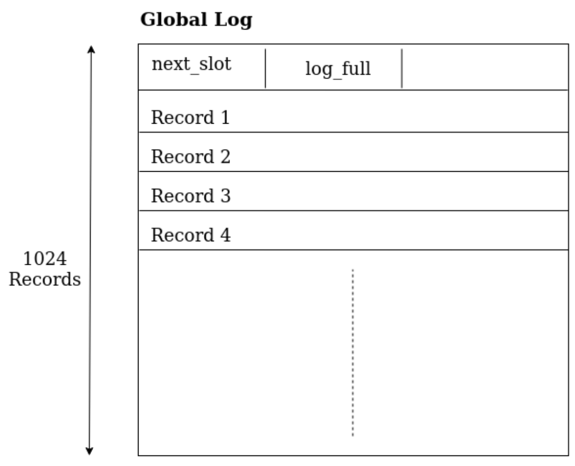
Function Pointer Overflow Detection

The main goals for this section are

1. Store function pointer value when they are declared in a reserved area.
2. Update the function pointer value when they are changed during normal execution of the process we are protecting
3. Check the stored value and current value when the process invokes a function call using the pointer.

To achieve these, we designed our following data structures.

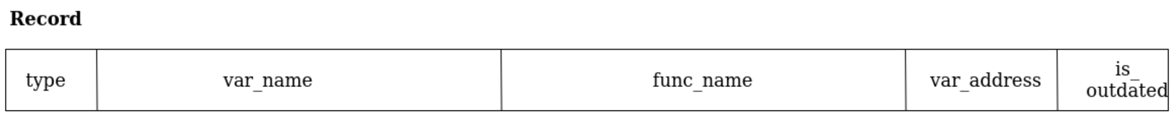
Data Structures and Supported Function Calls



Note: the source file can be found in "log.h"

This is the struct Global Log. Global Log stores maximum 1024 active records. The meaning of the variables are:

- next_slot: the next empty record slot
- log_full: indicate if the log was once full
- Record: each record stores one function pointer value and its associated information.



This is the Record Struct. Record is used for storing a single function pointer value. Global Log has 1024 Records space.

The meaning of the variables are:

- type: type of the variable (e.g. void, int)
- var_name: the name of the function pointer
- func_name: the name of function where this pointer is declared
- var_address: the value of function pointer
- is_outdated: if this function pointer is still valid

The supported function calls are:

- add_record: add one record, it calls all required setters
- set_next_slot: update the next available slot
- set_var_name: set the function pointer name
- set_var_address: set the function pointer value
- set_type: set the pointer type
- set_outdated: set records within a function call to be not all invalid/outdated. Whenever a function call from the process we are protecting is done execution, all the local function pointers we stored are no longer useful anymore, we therefore set out_dated_flags on those variables.
- update_value: update the value of a function pointer.

- check_value: check if the current value of the function pointer is the same as the stored one. If not exit the process.
- search_outdated_record: This function call is used when log is full once and it serves as a replacement algorithm. It searches for an outdated record and returns its slot number to its caller.
- print_log: print the elements in the log for debugging purposes.

We used a Python module: pycparser for C99 to obtain an AST tree from a preprocessed C99 file produced by cpp. After we obtained the AST. We will do a series of scanning and modification:

- When a function pointer is declared, add_record is added to the AST.
- When a function pointer is assigned to a new value during normal execution of the program: update_value is added to the AST
- When a function pointer is used: check_value is added to the AST before the pointer being used.
- Before a function call returns: set_outdated is added to the AST.

We then convert the modified AST tree back to C99 code using pycparser and re-start the compilation steps .

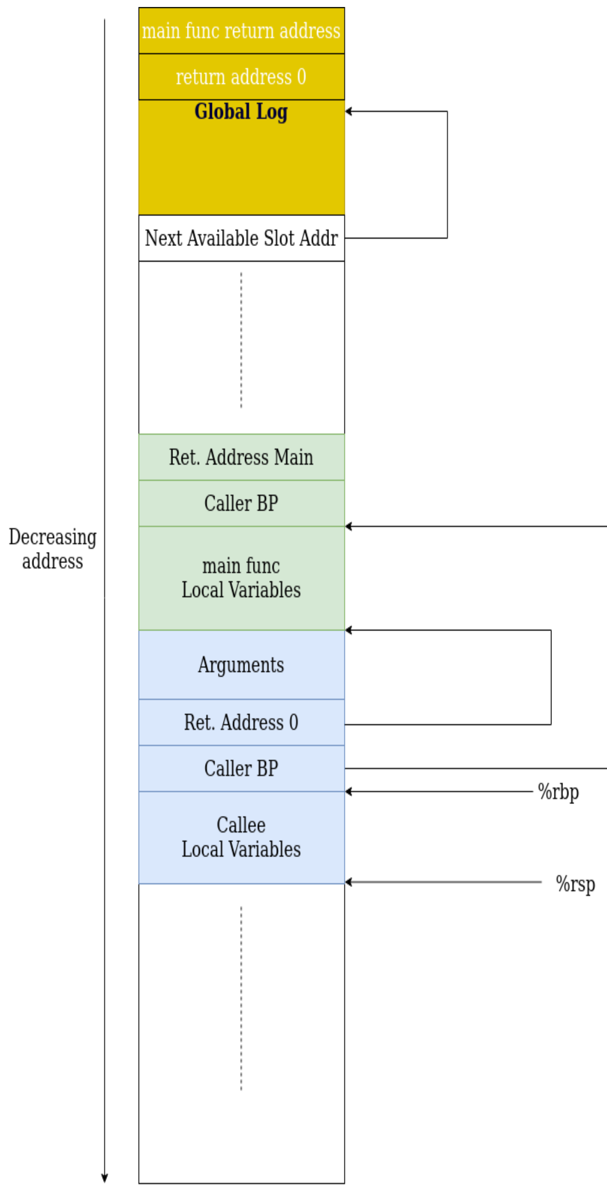
Integration with GCC-Plugin

At plugin_init, the source file is saved, then we invoke the python script for modifying AST and converting AST to C source code. Upon success the modified source file is then passed to GCC for further processing. A callback function is registered at event PLUGIN_FINISH, this callback function will roll back the changes we made to the source file and restore the source file to its original state.

Return Address Overwrite Detection & Recovery

In this section we will show the possible designs we came up with to address return address overwrites in buffer overflow. Two of the designs we came up with have a reserved area called Global Log. Though same name, this Global Log is not the same as the Global Log for function pointer overflow. Global Log referred in this section is an reserved 1024*8 byte space, it is not a user defined struct.

Candidate Design 1



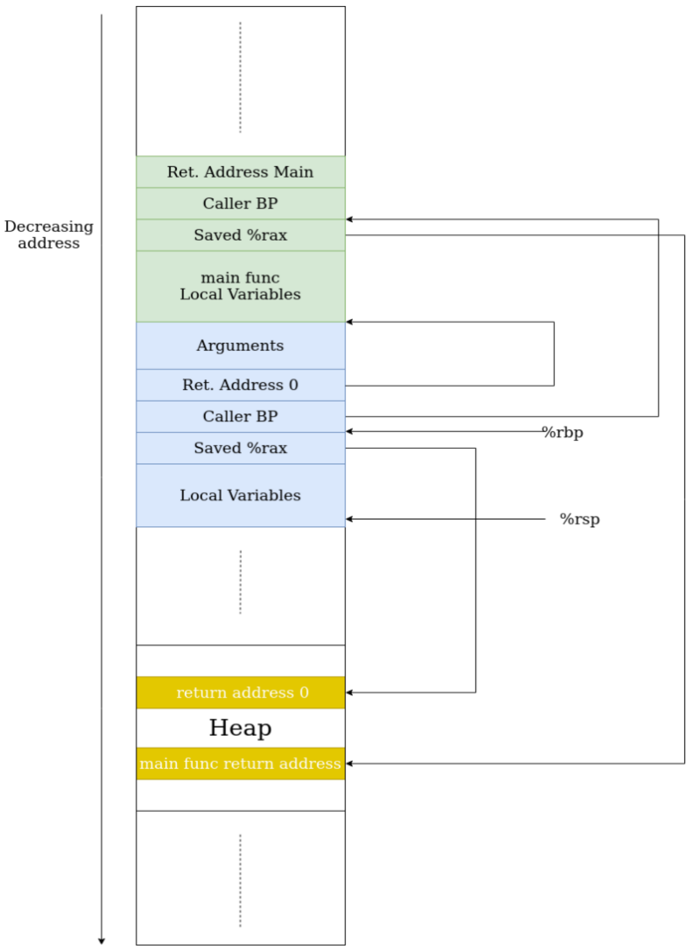
In Design 1, a program (denote P1) is ran to reserve 1025*8 bytes space on stack. The content of the 1025th slot is the address of the next available slot. Then when GCC-Plugin starts, we pass the addr = &(Next Available Slot Addr) to GCC- Plugin, so that GCC-Plugin knows where the next available slot is by accessing memory of addr.
Let denote the process we are protecting as P2.

At function prologue, the return address of every function in P2 is added to the next available slot, and the next available slot address is subtracted by 8.

At function epilogue, the return address is checked against the memory from the slot before the next available slot, and if there is an unmatch we call abort, if not we add 8 to the next available slot address.

The problem of this design is we have to keep P1 running until P2 is done execution, and we also need P1 to send &(Next Available Slot Addr) to GCC- Plugin, P2 will also have to signal P1 on its completion. This level of inter process communication is way too complex, we therefore did not go with this approach.

Candidate Design 2

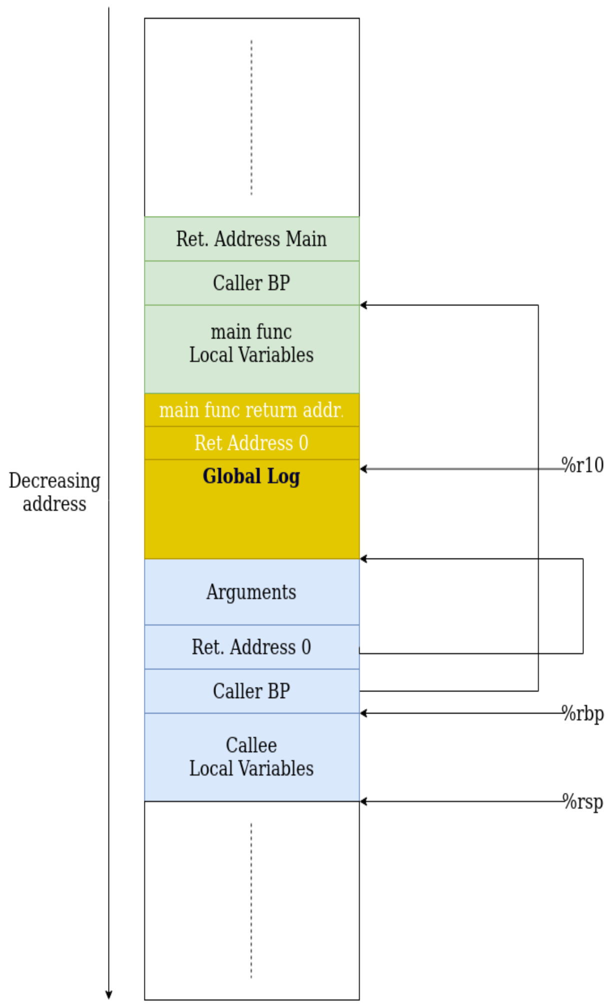


In Design 2, let’s denote the process we are protecting as P. At each function prologue of P, we call malloc to allocate us a space on heap, and we will save the value returned by malloc on stack by pushing %rax., then the pushed %rax value is always fixed at -8(%rbp) for each function call.

At each function epilogue of P, the return address is checked by accessing heap memory using the same %rax pointer value on stack, and if there is an unmatch we call abort, if not deallocate the heap memory.

Problem with this approach is that the save %rax resides too close to the return address, it is highly likely to be corrupted by an overflown buffer. On top of that, there is memory allocation and deallocation, which adds an extra layer of complexity. We therefore did not got with this design.

Candidate Design 3



In Design 3, let's denote the process we are protecting as P. A Global Log is reserved right after the main function's local variable declarations. Register %r10 is used to store the next available slot address. After doing lots of research, we found that gcc uses %r10 only as part of its trampoline for function pointers to GNU C nested functions. Since our program is not aimed for complex chain of function calls, we find it safe to use %r10.

At main function prologue of P, %rsp is subtracted by 1024*8, and the address of the first slot is stored to %r10.

At other function prologue of P, its return address is saved to where %r10 points to, %r10 is then subtracted by 8.
At main function epilogue of P, %rsp is added by 1024*8.

At other function epilogue of P, its return address is checked against where 8(%r10) points to, if there is a mismatch, abort is called. If not, increase %r10 by 8.

Design 3 is what we picked for it has much less overhead and much simpler to implement than Design 1 and more secure than Design 2.

In the next section, we are presenting how we achieved such design by using GCC Plugin RTL library.

RTL passes Setup in GCC plugin

- Structure of definition for a RTL pass

We use RTL passes to interact with the GCC compiler when the RTL representation is generated. The structure of the definition of RTL pass is shown below.

```
const pass_data my_pass_data =
{
    .type = RTL_PASS,
    .name = "ra_check",
    ...
};
struct my_pass : rtl_opt_pass
{
public:
    my_pass(gcc::context *ctx) : rtl_opt_pass(my_pass_data,ctx) {}
    virtual unsigned int execute(function* fun) override {...}
    virtual bool gate(function* fun) override {...}
    virtual my_pass* clone() override {return this; }
};

static struct register_pass_info my_pass_info =
{
    .pass = new my_pass(g),
    .reference_pass_name = "pro_and_epilogue",
    .ref_pass_instance_number = 0,
    .pos_op = PASS_POS_INSERT_AFTER,
};
```

Note we set `pass_data.type` to "`rtl`" and set `register_pass_info.reference_pass_name` to "`pro_and_epilogue`". This is the reason why we choose `rtl` pass to implement our design. For higher level passes like GIMPLE pass, we cannot locate the position of function prologue and epilogue.

- Parsing RTL representations: insn

After the structure is done, our main concern is how to parse the RTL codes and call the functions we want when it comes to the prologue and epilogue. This is where insn comes in. The RTL representation of the code for a function is a doubly-linked chain of objects called insns. We get an initial insn when the function comes in and keep looking for the next insn. For each insn there is NOTE to represent additional debugging and declarative information. Some examples are `NOTE_INSN_FUNCTION_BEG`, `NOTE_INSN_BLOCK_BEG`. The 2 notes we will use here is `NOTE_INSN_PROLOGUE_END` and `NOTE_INSN_EPILOGUE_BEG`. As the names suggest, the former denotes the end of function prologue and the latter denotes the beginning of function epilogue. We can then find where we need to insert our codes by simply parsing the insn chain. The partial code is shown below:

```
for (insn=get_insns(); insn; insn=NEXT_INSN(insn))
```

```
{
    if (NOTE_P(insn) && (NOTE_KIND(insn) == NOTE_INSN_PROLOGUE_END)) {...}
    else if (NOTE_P(insn) && (NOTE_KIND(insn) == NOTE_INSN_EPILOGUE_BEG)) {...}
}
```

From Assembly to RTL

To write the RTL expressions, we need to figure out what assembly codes to insert and where to insert them first. Let's gather the operations needed first. In the prologue of main(), we reserve a space for the Global Log and use a register (%r10) as a pointer to the next available slot for a new address. In prologue of functions, we buffer the return address of the function in the stack space pointed by %r10. In epilogue of function, we compare the current return address with the buffered one. If there is a mismatch, which means the return address is overwritten, we can either restore the return address or abort the program. We choose to restore the address here for an obvious result. And lastly in main() epilogue, we restore (%rsp) since we have decremented it for space for Global Log.

And here is the Assembly codes we need for this purpose:

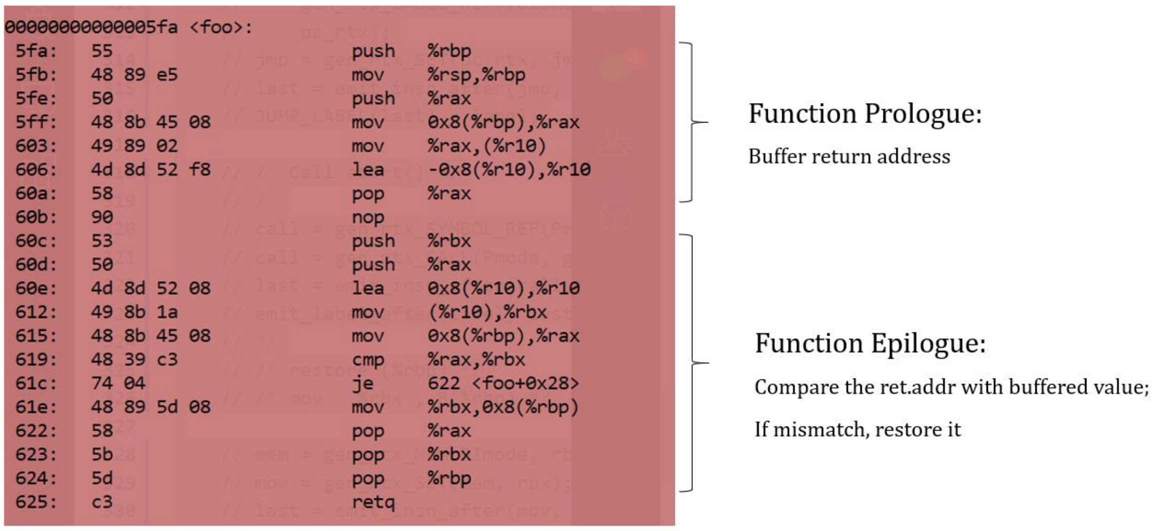
```
In main() prologue:
         subq    $8192, %rsp              # void* ret_addr[1024]
         leaq    8184(%rsp), %r10         # use register %r10 to store the next
                                          # available slot
In other functions prologue:
         pushq   %rax                     # save %rax value
         movq    8(%rbp), %rax            # move ret. address to %rax
         movq    %rax, (%r10)             # assign next_slot the return address,
                                          # which is the memory location where
                                          # %r10 points to
         addq    $8, %r10                 # now next_slot is ret_addrr[1]
         popq    %rax                     # restore the value of %rax
In other functions epilogue:
         add     $8, %r10                 # let %r10 points to the value we want
         push    %rbx, %rcx
         mov     (%r10), %rbx             # %rbx = addr1 (buffered ret.addr)
         mov     8(%rbp), %rcx            # %rcx = addr2 (current ret.addr)
         cmp     %rbx, %rcx               # compare the 2 values
         jeq     [label]                  # skip the next instruction if equal
         mov     %rbx, 8(%rbp)            # restore ret.addr
[label] ⇒pop     %rcx, %rbx                # restore %rbx, %rcx
In main epilogue:
        addq     $8192, %rsp              # decrement %rsp
```

Now the task is to turn the above Assembly codes to RTL expressions. After a great amount time for consulting relevant materials, we find there are several useful RTX functions:

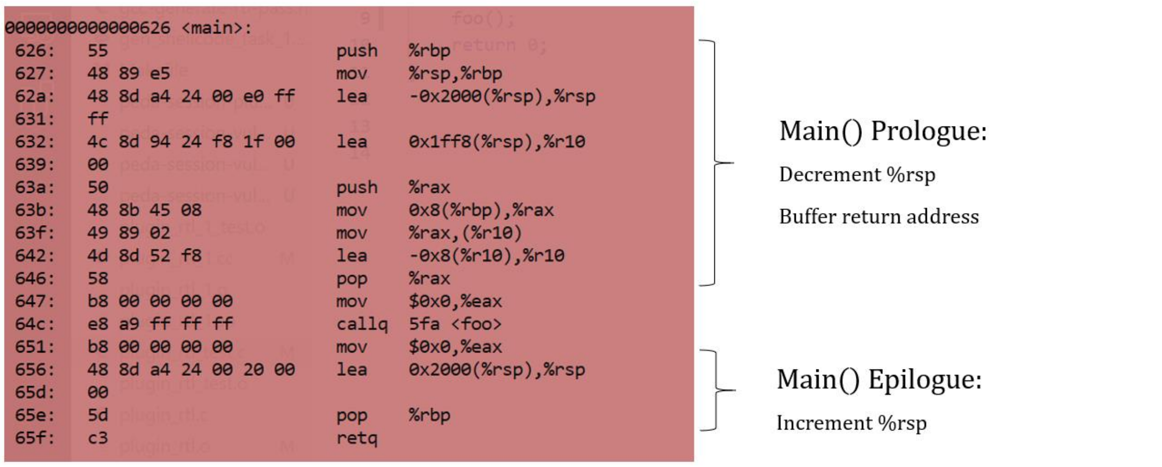
Examples:

```
rtx gen_rtx_REG(Mode, Reg No.);  # access a register
rtx gen_rtx_MEM(Mode, Address);  # access memory in stack by the address
rtx gen_rtx_SET(Dest, Source);   # set the value of dest. to the value of
                                 # source, similar to ‘mov’
```

Then we finally finish this part. The result of inserted Assembly codes is shown below:



```
00000000000005fa <foo>:
 5fa:   55                      push   %rbp
 5fb:   48 89 e5                mov    %rsp,%rbp
 5fe:   50                      push   %rax
 5ff:   48 8b 45 08             mov    0x8(%rbp),%rax
 603:   49 89 02                mov    %rax,(%r10)
 606:   4d 8d 52 f8             lea    -0x8(%r10),%r10
 60a:   58                      pop    %rax
 60b:   90                      nop
 60c:   53                      push   %rbx
 60d:   50                      push   %rax
 60e:   4d 8d 52 08             lea    0x8(%r10),%r10
 612:   49 8b 1a                mov    (%r10),%rbx
 615:   48 8b 45 08             mov    0x8(%rbp),%rax
 619:   48 39 c3                cmp    %rax,%rbx
 61c:   74 04                   je     622 <foo+0x28>
 61e:   48 89 5d 08             mov    %rbx,0x8(%rbp)
 622:   58                      pop    %rax
 623:   5b                      pop    %rbx
 624:   5d                      pop    %rbp
 625:   c3                      retq
```



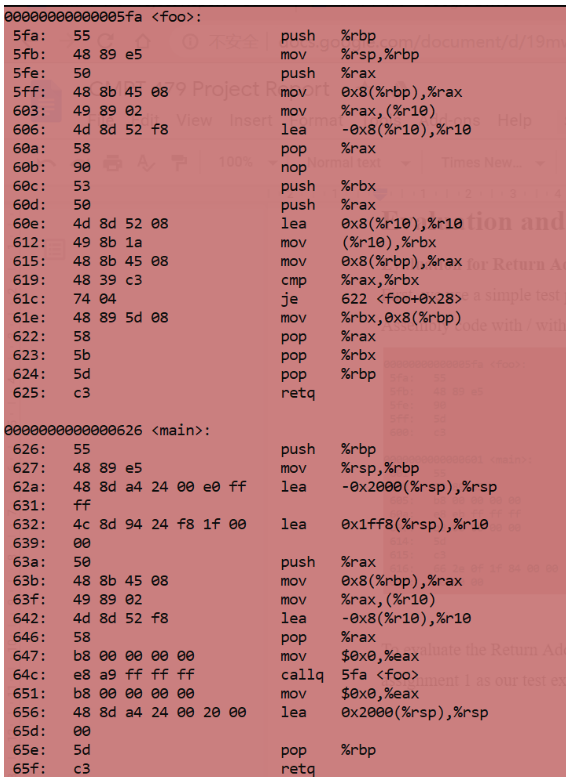
```
0000000000000626 <main>:
 626:   55                      push   %rbp
 627:   48 89 e5                mov    %rsp,%rbp
 62a:   48 8d a4 24 00 e0 ff    lea    -0x2000(%rsp),%rsp
 631:   ff
 632:   4c 8d 94 24 f8 1f 00    lea    0x1ff8(%rsp),%r10
 639:   00
 63a:   50                      push   %rax
 63b:   48 8b 45 08             mov    0x8(%rbp),%rax
 63f:   49 89 02                mov    %rax,(%r10)
 642:   4d 8d 52 f8             lea    -0x8(%r10),%r10
 646:   58                      pop    %rax
 647:   b8 00 00 00 00          mov    $0x0,%eax
 64c:   e8 a9 ff ff ff          callq  5fa <foo>
 651:   b8 00 00 00 00          mov    $0x0,%eax
 656:   48 8d a4 24 00 20 00    lea    0x2000(%rsp),%rsp
 65d:   00
 65e:   5d                      pop    %rbp
 65f:   c3                      retq
```

In the above pictures, we only store and check for the return addresses. How, we actually buffer and check for **both Return Addresses and Base Pointers** to tackle a bug that occurs during the test process.

Evaluation and Experimentation

Evaluation for Return Address Overwrite Detection & Recovery

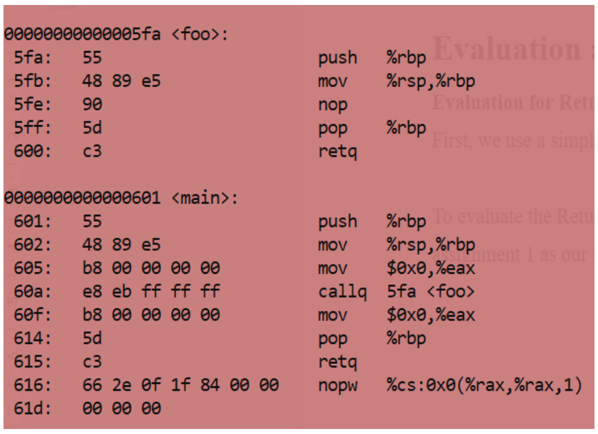
First, we use a simple test program to check if the Assembly codes are correctly inserted. Below is the Assembly code with / without our plugin.



```
00000000000005fa <foo>:
 5fa:   55                      push   %rbp
 5fb:   48 89 e5                mov    %rsp,%rbp
 5fe:   50                      push   %rax
 5ff:   48 8b 45 08             mov    0x8(%rbp),%rax
 603:   49 89 02                mov    %rax,(%r10)
 606:   4d 8d 52 f8             lea    -0x8(%r10),%r10
 60a:   58                      pop    %rax
 60b:   90                      nop
 60c:   53                      push   %rbx
 60d:   50                      push   %rax
 60e:   4d 8d 52 08             lea    0x8(%r10),%r10
 612:   49 8b 1a                mov    (%r10),%rbx
 615:   48 8b 45 08             mov    0x8(%rbp),%rax
 619:   48 39 c3                cmp    %rax,%rbx
 61c:   74 04                   je     622 <foo+0x28>
 61e:   48 89 5d 08             mov    %rbx,0x8(%rbp)
 622:   58                      pop    %rax
 623:   5b                      pop    %rbx
 624:   5d                      pop    %rbp
 625:   c3                      retq

0000000000000626 <main>:
 626:   55                      push   %rbp
 627:   48 89 e5                mov    %rsp,%rbp
 62a:   48 8d a4 24 00 e0 ff    lea    -0x2000(%rsp),%rsp
 631:   ff
 632:   4c 8d 94 24 f8 1f 00    lea    0x1ff8(%rsp),%r10
 639:   00
 63a:   50                      push   %rax
 63b:   48 8b 45 08             mov    0x8(%rbp),%rax
 63f:   49 89 02                mov    %rax,(%r10)
 642:   4d 8d 52 f8             lea    -0x8(%r10),%r10
 646:   58                      pop    %rax
 647:   b8 00 00 00 00          mov    $0x0,%eax
 64c:   e8 a9 ff ff ff          callq  5fa <foo>
 651:   b8 00 00 00 00          mov    $0x0,%eax
 656:   48 8d a4 24 00 20 00    lea    0x2000(%rsp),%rsp
 65d:   00
 65e:   5d                      pop    %rbp
 65f:   c3                      retq
```

b) with plugin



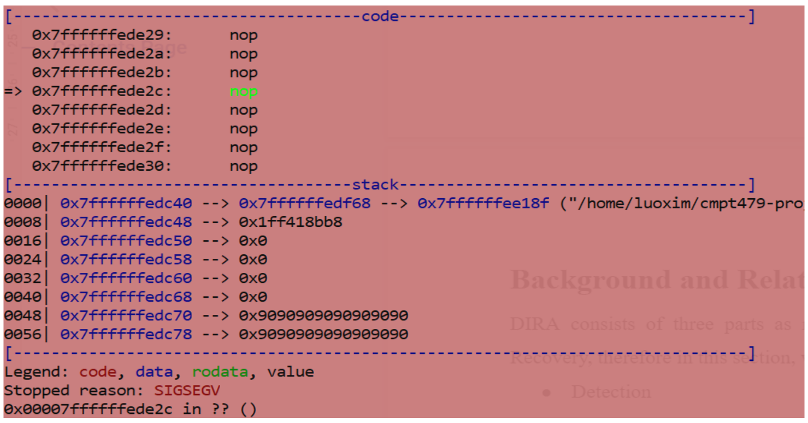
```
00000000000005fa <foo>:
 5fa:   55                      push   %rbp
 5fb:   48 89 e5                mov    %rsp,%rbp
 5fe:   90                      nop
 5ff:   5d                      pop    %rbp
 600:   c3                      retq

0000000000000601 <main>:
 601:   55                      push   %rbp
 602:   48 89 e5                mov    %rsp,%rbp
 605:   b8 00 00 00 00          mov    $0x0,%eax
 60a:   e8 eb ff ff ff          callq  5fa <foo>
 60f:   b8 00 00 00 00          mov    $0x0,%eax
 614:   5d                      pop    %rbp
 615:   c3                      retq
 616:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
 61d:   00 00 00
```

b) without plugin

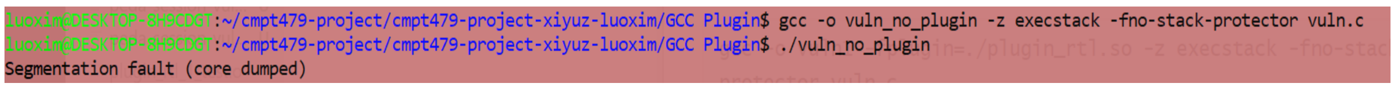
From the comparison of these 2 different Assembly codes, we can easily see how the plugin works. It inserts codes in function prologue and epilogue.

Second, in order to evaluate if the plugin really works in detecting and repairing the victim program, we use the vulnerable program in assignment 1 as our test example. Unfortunately, I did not manage to spawn a shell on my computer, which is in 64-bit. But I do make the return address jump to NOP sled before the shell code, shown as below.



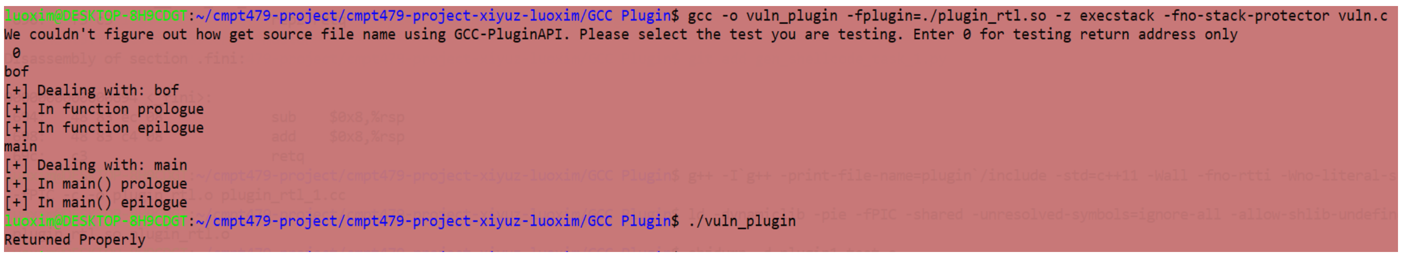
```
[------------------------------------code-------------------------------------]
   0x7ffffffede29:      nop
   0x7ffffffede2a:      nop
   0x7ffffffede2b:      nop
=> 0x7ffffffede2c:      nop
   0x7ffffffede2d:      nop
   0x7ffffffede2e:      nop
   0x7ffffffede2f:      nop
   0x7ffffffede30:      nop
[------------------------------------stack------------------------------------]
0000| 0x7ffffffedc40 --> 0x7ffffffedf68 --> 0x7ffffffee18f ("/home/luoxim/cmpt479-pro
0008| 0x7ffffffedc48 --> 0x1ff418bb8
0016| 0x7ffffffedc50 --> 0x0
0024| 0x7ffffffedc58 --> 0x0
0032| 0x7ffffffedc60 --> 0x0
0040| 0x7ffffffedc68 --> 0x0
0048| 0x7ffffffedc70 --> 0x9090909090909090
0056| 0x7ffffffedc78 --> 0x9090909090909090
[-----------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x00007ffffffede2c in ?? ()
```

And if I just compile vuln.c as I did in assignment 1, I will get segmentation fault.



```
luoxim@DESKTOP-8H9CDGT:~/cmpt479-project/cmpt479-project-xiyuz-luoxim/GCC Plugin$ gcc -o vuln_no_plugin -z execstack -fno-stack-protector vuln.c
luoxim@DESKTOP-8H9CDGT:~/cmpt479-project/cmpt479-project-xiyuz-luoxim/GCC Plugin$ ./vuln_no_plugin
Segmentation fault (core dumped)
```

But if I compile it with our plugin, it will get the output “Returned Properly”. (The last statement before return in main()).



```
luoxim@DESKTOP-8H9CDGT:~/cmpt479-project/cmpt479-project-xiyuz-luoxim/GCC Plugin$ gcc -o vuln_plugin -fplugin=./plugin_rtl.so -z execstack -fno-stack-protector vuln.c
We couldn't figure out how get source file name using GCC-PluginAPI. Please select the test you are testing. Enter 0 for testing return address only
 0
bof
[+] Dealing with: bof
[+] In function prologue
[+] In function epilogue
main
[+] Dealing with: main
[+] In main() prologue
[+] In main() epilogue
luoxim@DESKTOP-8H9CDGT:~/cmpt479-project/cmpt479-project-xiyuz-luoxim/GCC Plugin$ ./vuln_plugin
Returned Properly
```

From the above observation, the plugin does the right thing, avoiding the program run into the NOP sled, which will result in spawning a shell and let the attacker take control of the process.

Problem with Our Program

- We tried our best to address the problem, yet our program is still way too simple to be used in real life. For example, in the function pointer overflow detection part, our program cannot handle function pointer passing as parameters as it would require more static analysis to make control flow graph, and we did not have enough time to complete it. Our program also does not support multi threading, as the log is global and we assumed unique variable definition, yet in multi threading, each thread has its own local variables copy.
- Due to GCC Plugin API problem, we were not able to get the source file that our plugin has an effect on. That means when testing, we have to manually add the test file name to plugin_init.
- There are lots of test cases to be considered and what we have done are the most basic ones. There are still some issues that we keep thinking about. For example, what is the best place for allocating to Global Log in RTL implementation. Our current approach set it in the stack space for main() function. However, this may results in many problems, like the check the return address of main(). More seriously, in our implementation, we have to use a global register, %r10 as the pointer to the next available slot. It won’t work if some function needs this register (though commonly it will not happen). How about using the heap to store the Global Log? There are many unsolved problems like this.

Tesing Procedures

Please follow the README file for testing.

Unfortunately, we were not able to develop a successful test that adds both function pointer overflow protection and return address overflow protection. Such tests result in segmentation fault, and we were unsure of the cause. We therefore will separate the two tests. One for function pointer and the other for return address.

Our tests outcomes are positive, the details will be shown in the video we provide.

Learned Lessons

- We change our original plan of implementing the dynamic recovery mechanism in a function call tree. We made attempts to fulfill that in multiple ways, like drawing the CFG graph, recovery the return address. However, the recovery in DIRA is much complex than what we have done. This is because the limited time and knowledge basis.
- It is extremely difficult as we are not familiar with x86-64 assembly language and usually for lower end development
- lack of documentation is another barrier we had to overcome. We heavily relied on reading source codes and outdated tutorials. The different API across versions is a problem that was also very time-consuming to tackle . However, we do believe that our knowledge base has been expanded. We are now familiar with execution steps, memory layout and our ability in reading/writing assembly code has been increased significantly.
- Time schedule needs to be more realistic. Our original plan is to complete a demo before the milestone presentation. However, it looks quite naive now. So, we plan to make another timeline after today's class.

Conclusion

To conclude, though we proposed to do both detection and recovery, after careful discussion, we decided to give up on the recovery part as it is way more complicated than we initially thought. Overall, our program is functioning but far away from being complete.

References

[1] Smirnov, A., & Chiueh, T. C. DIRA: Automatic Detection, Identification, and Repair of Control-Hijacking Attacks.
[2] David Evans & David Larochelle (2002), University of Virginia. Improving Security Using Extensible Lightweight Static Analysis.
[3] Wei Le & Mary Lou Soffa (2008), University of Virginia. Marple: A Demand-Driven Path-Sensitive Buffer Overflow Detector
[4] GCC Internal. Retrieved from https://gcc.gnu.org/onlinedocs/gccint/index.html#Top
[4] Roger Ferrer Ibáñez.(2015, August 16) A simple plugin for GCC [Blog post]. Retrieved from https://thinkingeek.com/2015/08/16/simple-plugin-gcc-part-2/