Taint analysis and pattern matching with Pin

by Jonathan Salwan (http://twitter.com/JonathanSalwan) - 2013-08-08

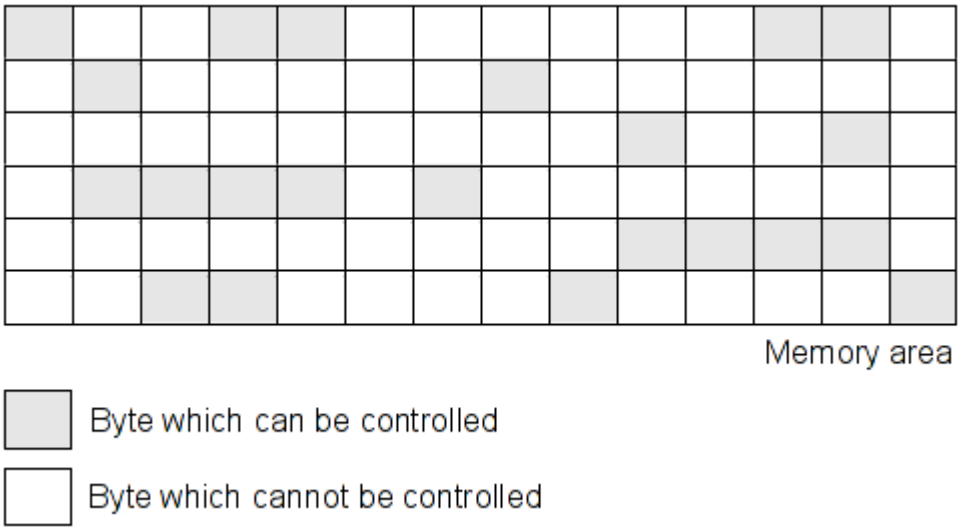
Last weeks I played with the Pin API and this post can be considered as my personal notepad. All examples written in this post are just proof of concept thus not 100% reliable. But it can maybe give some ideas for other people.

Edit 2015-11: Check out our Pin-based framework (http://triton.quarkslab.com/) which offers the possibility to deal with a taint engine using Python bindings.

1 - Introduction

1.1 - Concept

The taint analysis is a popular method which consists to check which variables can be modified by the user input. All user input can be dangerous if they aren't properly checked. With this method it is possible to check the registers and the memory areas which can be controlled by the user when a crash occurs - That can be useful.



In order to know if an area is readable/writeable is very straightforward. The difficulty is to check if this area can be controlled by the user and to spread the taints. For example see the following code.

```
/* Example 1 */
void foo1(const char *av[])
{
  uint32_t a, b;

  a = atoi(av[1]);
  b = a;
  foo2(b);
}

/* Example 2 */
void foo2(const char *av[])
{
  uint8_t *buffer;

  if (!(buffer = (uint8_t *)malloc(32 * sizeof(uint8_t))))
    return(-ENOMEM);

  buffer[2]  = av[1][4];
  buffer[12] = av[1][8];
  buffer[30] = av[1][12];
}
```

In the first example, at the beginning, the '**a**' and '**b**' variables are not tainted. When the **atoi** function is called the '**a**' variable is tainted. Then '**b**' is tainted when assigned by the '**a**' value. Now we know that the **foo2** function argument can be controlled by the user.

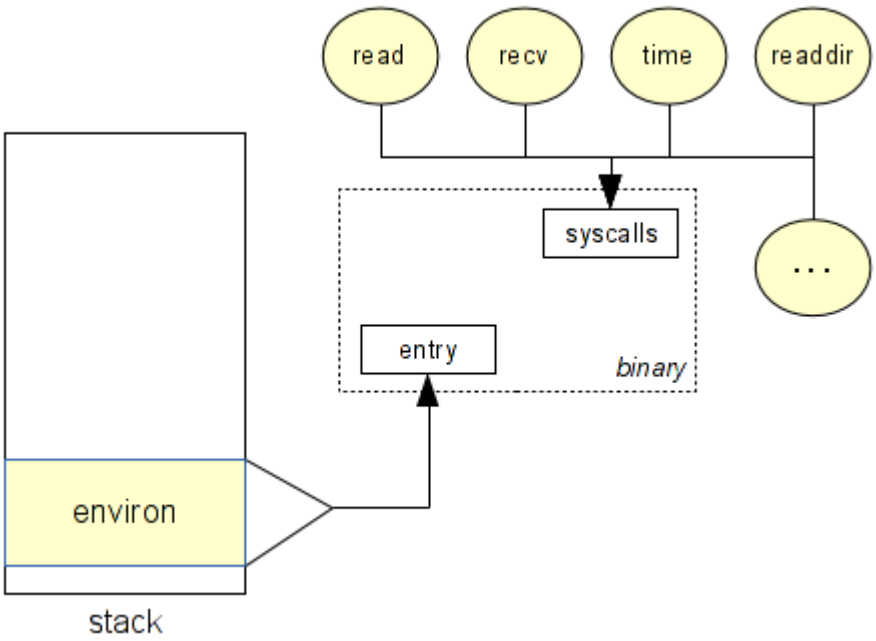
In the second example, when the buffer is allocated via **malloc** the content is not tainted. Then when the allocated area is initiazlied by user inputs, we need to taint the bytes '**buffer+2**', '**buffer+12**' and '**buffer+30**'. Later, when one of those bytes is read, we know it can be controlled by the user.

1.2 - How taint

We have two possible ways, static or dynamic analysis. For each of them, we can find some pros and cons.

1.2.1 - Dynamic analysis

For the dynamic analysis, basically we need to determinate all user inputs like environment and syscalls. We begin to taint these inputs and we spread/remove the taint when we have instructions like GET/PUT, LOAD/STORE.



In order to do this, we need a dynamic binary instrumentation framework. The purpose of the DBI is to add a pre/post handler on each instruction. When a handler is called, you are able to retrieve all the information you want about the instruction or the environment (memory).

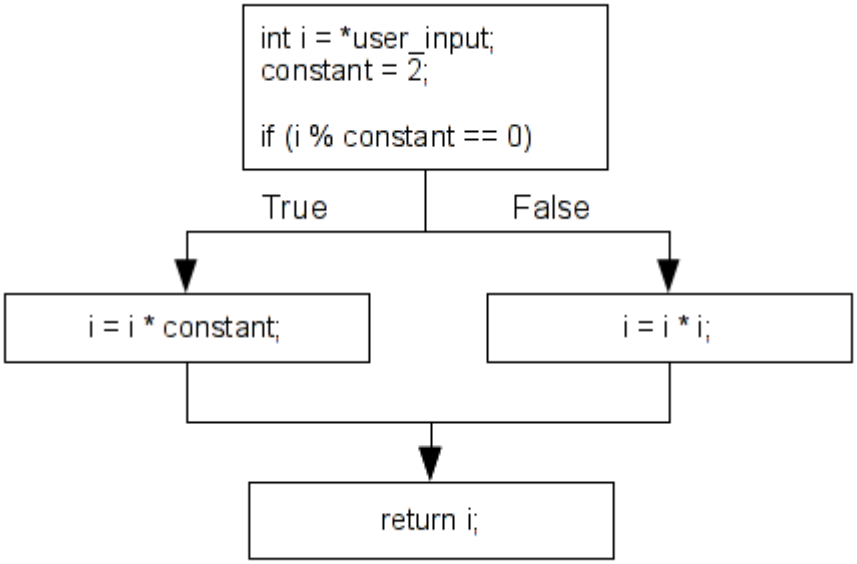
Several tools provide an intermediate representation (IR). For example, Valgrind (http://valgrind.org/) is a popular instrumentation framework which uses an IR (Vex). Generally with an IR, each variable is on SSA-based (http://en.wikipedia.org/wiki/Static_single_assignment_form) form (Static Single Assignment), with that it is easier to taint and to manage your memory. To show you, an example about VEX and the SSA form, the following code is a Vex representaion to the `add eax, ebx` instruction.

```
t3 = GET:I32(0)     # get %eax, a 32-bit integer (t3 = eax)
t2 = GET:I32(12)    # get %ebx, a 32-bit integer (t2 = ebx)
t1 = Add32(t3,t2)   # eger (t2 = ebx)
PUT(0) = t1         put %eax (eax = t1)
```

I chose to use Pin (http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool): a C++ dynamic binary instrumentation framework (without IR) written by Intel.

1.2.2 - Static analysis

The advantage of using static analysis is the fact that it provides better code coverage than dynamic analysis. As you can see below, when the code is parsed, we can provide a CFG and detect all branchs.



On the other hand, the principal disadvantage of the static analysis is that it's not as accurate than the dynamic analysis - It cannot acces the runtime information for example. We can't retrieve registers or memory values.

1.2.3 - Dynamic or Static ?

As you can see, each approaches have some advantages and disadvantages... If you use dynamic analysis we can't cover all the code but you will be more reliable. If you use static analysis you can cover the code, but you can't get the context information at runtime. I have chosen to take the dynamic analysis and talk about the Pin usage. Maybe in a future blog post I will talk more about the static analysis - Static analysis is a very interesting method and we can do a lot of great things from the AST (Abstract Syntax Tree).

1.3 - Some problematics

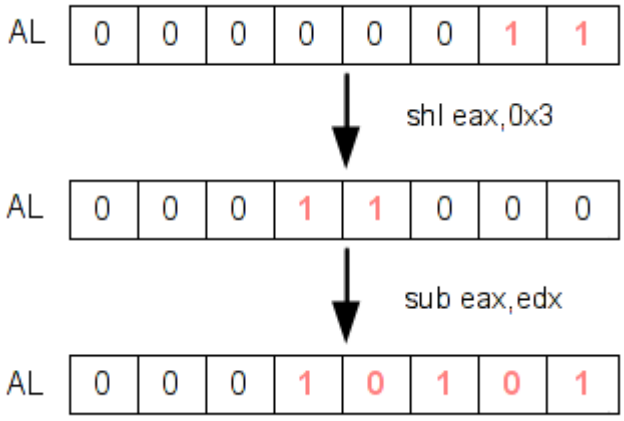
During research about this method, I encountered several problems. I think we can find lot of interesting problems about the taint analysis.

1.3.1 - Byte or bit ?

One of these problematic is to determine what method is the more accurate to do a taint with a great precision. For example, what are we supposed to do when a controlled byte is multiplied and stored somewhere in memory ? Should we taint the destination variable ? See the following code.

```
; uint32_t num, x;
;
; x = atoi(av[1])
; if (x > 0 && x < 4)
;   num = 7 * x;
; return num;

400556:   call   400440 <atoi@plt>
40055b:   mov    edx,eax
40055d:   mov    eax,edx
40055f:   shl    eax,0x3
400562:   sub    eax,edx
400564:   mov    DWORD PTR [rbp-0x4],eax
400567:   mov    eax,DWORD PTR [rbp-0x4]
40056a:   leave
40056b:   ret
```

In the previous code, we can control only 5 bits of the variable 'num' ; not the whole integer. So, we can't say that we control the totality of this variable when it is returned and used somewhere else.



So, what to do? Tainting bytes is easier and light or taining bits controlled by the user? If you taint bytes, it will be easier but not 100% reliable. If we taint bits, it will be harder and more difficult to manage the taint tree but it will be 99% reliable. That's one of several problems we have to solve!

2 - Simple taint a memory area

For this first example, we are going to taint the '**read**' memory area and we will see a brief overview of the Pin API. For this first test we will :

1. Catch the sys_read syscall.
2. Get the second and the third argument for taint area.
3. Call an handler when we have an instruction like LOAD or STORE in this area.

We will work on this following code, the function **foo**, does a simple **LOAD** and **STORE** instructions.

```
void foo(char *buf)
{
  char a;

  a = buf[0];
  a = buf[4];
  a = buf[8];
  a = buf[10];
  buf[5]  = 't';
  buf[10] = 'e';
  buf[20] = 's';
  buf[30] = 't';
}

int main(int ac, char **av)
{
  int fd;
  char *buf;

  if (!(buf = malloc(256)))
    return -1;

  fd = open("./file.txt", O_RDONLY);
  read(fd, buf, 256), close(fd);
  foo(buf);
}
```

2.1 - Catch the syscalls

With Pin it is possible to add a pre and post handler when a syscall occurs. For that, we just need to initialize the callback function.

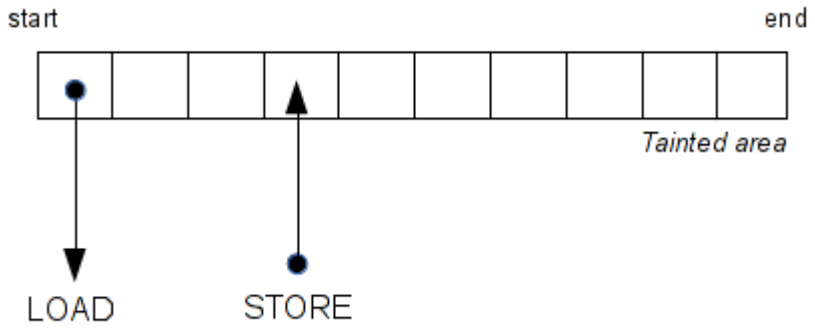
```
typedef VOID(* LEVEL_PINCLIENT::SYSCALL_ENTRY_CALLBACK)(THREADID threadIndex,
                                                        CONTEXT *ctxt,
                                                        SYSCALL_STANDARD std,
                                                        VOID *v);

typedef VOID(* LEVEL_PINCLIENT::SYSCALL_EXIT_CALLBACK)(THREADID threadIndex,
                                                       CONTEXT *ctxt,
                                                       SYSCALL_STANDARD std,
                                                       VOID *v);

VOID LEVEL_PINCLIENT::PIN_AddSyscallEntryFunction(SYSCALL_ENTRY_CALLBACK fun, VOID *val);
VOID LEVEL_PINCLIENT::PIN_AddSyscallExitFunction(SYSCALL_EXIT_CALLBACK fun, VOID *val);
```

In our case we will just use the `LEVEL_PINCLIENT::PIN_AddSyscallEntryFunction`. When a syscall occurs, we will check if the syscall is **read**. Then, we save the second and third argument which describe our memory area.

```
/* area of bytes tainted */
struct range
{
  UINT64 start;
  UINT64 end;
};

std::list<struct range> bytesTainted;

VOID Syscall_entry(THREADID thread_id, CONTEXT *ctx, SYSCALL_STANDARD std, void *v)
{
  struct range taint;

  /* If the syscall is read take the branch */
  if (PIN_GetSyscallNumber(ctx, std) == __NR_read){

      /* Get the second argument */
      taint.start = static_cast<UINT64>((PIN_GetSyscallArgument(ctx, std, 1)));

      /* Get the third argument */
      taint.end   = taint.start + static_cast<UINT64>((PIN_GetSyscallArgument(ctx, std, 2)));

      /* Add this area in our tainted bytes list */
      bytesTainted.push_back(taint);

      /* Just display information */
      std::cout << "[TAINT]\t\t\tbytes tainted from " << std::hex << "0x" << taint.start \
      << " to 0x" << taint.end << " (via read)"<< std::endl;
  }
}

int main(int argc, char *argv[])
{
    /* Init Pin arguments */
    if(PIN_Init(argc, argv)){
        return Usage();
    }

    /* Add the syscall handler */
    PIN_AddSyscallEntryFunction(Syscall_entry, 0);

    /* Start the program */
    PIN_StartProgram();

    return 0;
}
```

2.2 - Catch the LOAD and STORE instructions

Now we need to cach all instructions that read (**LOAD**) or write (**STORE**) in the tainted area. To do that, we will add a function called each time an access to this area is made.



For that we will add a master handler called for each instruction.

```
typedef VOID(* LEVEL_PINCLIENT::INS_INSTRUMENT_CALLBACK)(INS ins, VOID *v);
VOID LEVEL_PINCLIENT::INS_AddInstrumentFunction(INS_INSTRUMENT_CALLBACK fun, VOID *val);
```

Then, in the master handler we need to find the **LOAD** / **STORE** instruction, for example, **mov rax, [rbx]** or **mov [rbx], rax**.

```
if (INS_MemoryOperandIsRead(ins, 0) && INS_OperandIsReg(ins, 0)){
  INS_InsertCall(
      ins, IPOINT_BEFORE, (AFUNPTR)ReadMem,
      IARG_ADDRINT, INS_Address(ins),
      IARG_PTR, new string(INS_Disassemble(ins)),
      IARG_MEMORYOP_EA, 0,
      IARG_END);
}
else if (INS_MemoryOperandIsWritten(ins, 0)){
  INS_InsertCall(
      ins, IPOINT_BEFORE, (AFUNPTR)WriteMem,
      IARG_ADDRINT, INS_Address(ins),
      IARG_PTR, new string(INS_Disassemble(ins)),
      IARG_MEMORYOP_EA, 0,
      IARG_END);
}

int main(int argc, char *argv[])
{
  ...
  INS_AddInstrumentFunction(Instruction, 0);
  ...
}
```

As you can see above, we make some checks before insert a call. If the instruction's second operand read in the memory and if the first operand is a register. That means if the instruction looks like '**mov reg, [r/imm]**', it calls the ReadMem function. When calling this function, it passes several information like: the instruction address, the disassembly, and the address of the memory read. Same thing with the **STORE** instructions.

Now we just need to check if the accessed memory is in the tainted area.. For our memory read callback we have something like that:

```
VOID ReadMem(UINT64 insAddr, std::string insDis, UINT64 memOp)
{
  list<struct range>::iterator i;
  UINT64 addr = memOp;

  for(i = bytesTainted.begin(); i != bytesTainted.end(); ++i){
      if (addr >= i->start && addr < i->end){
        std::cout << std::hex << "[READ in " << addr << "]\t" << insAddr
          << ": " << insDis<< std::endl;
      }
  }
}
```

2.3 - Output

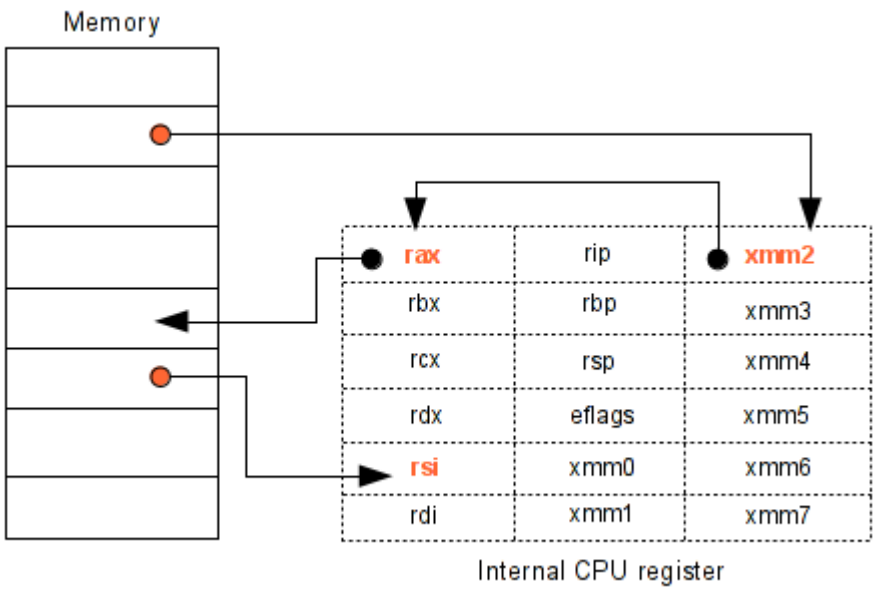
As you can see below, the Pin tool taints the read memory and display all **LOAD**/**STORE** from our area.

```
$ ../../../pin -t ./obj-intel64/Taint_ex1.so -- ./test_ex1
[TAINT]             bytes tainted from 0x665010 to 0x665110 (via read)
[READ in 665010]    400620: movzx eax, byte ptr [rax]
[READ in 665014]    40062a: movzx eax, byte ptr [rax+0x4]
[READ in 665018]    400635: movzx eax, byte ptr [rax+0x8]
[READ in 66501a]    400640: movzx eax, byte ptr [rax+0xa]
[WRITE in 665015]   40064f: mov byte ptr [rax], 0x74
[WRITE in 66501a]   40065a: mov byte ptr [rax], 0x65
[WRITE in 665024]   400665: mov byte ptr [rax], 0x73
[WRITE in 66502e]   400670: mov byte ptr [rax], 0x74
```

You can see the source code of this example 1 here : example 1 (./taint_example_1.cpp).

3 - Spread the taint

Detecting the access made is the tainted memory area is cool, but it's not enough. Imagine you **LOAD** a value in a register from the tainted memory, then you **STORE** this register in another memory location. In this case, we need to taint the register and the new memory location. Same way, if a constant is **STORED** in the memory area tainted, we need to delete the taint because the user can't control this memory location anymore.



3.1 - Requirement

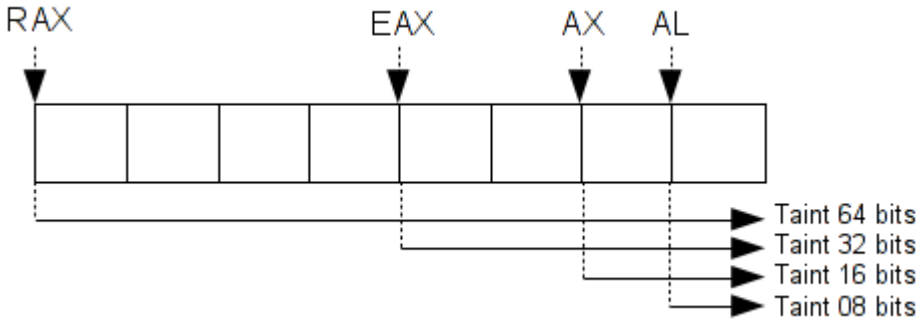
Based on the previous Pin tool, we modified it to spread the tainted memory. First, we have changed the memory area structure - `struct range`. Instead of tainting a memory range, we taint a specific unique address. Now we have a **std::list** storing all address tainted.

```
std::list<UINT64> addressTainted;
```

To spread the taint in register, we have added a **std::list** of **REG**. This list contains all registers controlled by the user input.

```
std::list<REG> regsTainted;
```

For each register, Pin assigns a unique enum. That why when you taint a register you need to taint the smaller size registers.



For that, we can use a big switch case which taints the target register and their all smaller size registers. Same way when you need to delete a tainted register.

```
switch(reg){

    case REG_RAX:  regsTainted.push_front(REG_RAX);
    case REG_EAX:  regsTainted.push_front(REG_EAX);
    case REG_AX:   regsTainted.push_front(REG_AX);
    case REG_AH:   regsTainted.push_front(REG_AH);
    case REG_AL:   regsTainted.push_front(REG_AL);
         break;

    /* ... */
```

3.2 - Memory spread

To spread the tainted memory, we just need to change our readMem and writeMem function.

```
VOID ReadMem(UINT64 insAddr, std::string insDis, UINT32 opCount, REG reg_r, UINT64 memOp)
{
  list<UINT64>::iterator i;
  UINT64 addr = memOp;

  if (opCount != 2)
    return;

  for(i = addressTainted.begin(); i != addressTainted.end(); i++){
      if (addr == *i){
        std::cout << std::hex << "[READ in " << addr << "]\t" << insAddr << ": "
          << insDis << std::endl;
        taintReg(reg_r);
        return ;
      }
  }
  /* if mem != tained and reg == taint => free the reg */
  if (checkAlreadyRegTainted(reg_r)){
    std::cout << std::hex << "[READ in " << addr << "]\t" << insAddr << ": "
      << insDis << std::endl;
    removeRegTainted(reg_r);
  }
}
```

As you can see above, when the program loads a value from the tainted area, we check if this memory location is tainted. If it is true, we taint the destination register. Otherwise, the memory is not tainted, so we check if the destination register is tainted. If not, we remove the register because we can't control the memory location.

```
VOID WriteMem(UINT64 insAddr, std::string insDis, UINT32 opCount, REG reg_r, UINT64 memOp)
{
  list<UINT64>::iterator i;
  UINT64 addr = memOp;

  if (opCount != 2)
    return;

  for(i = addressTainted.begin(); i != addressTainted.end(); i++){
      if (addr == *i){
        std::cout << std::hex << "[WRITE in " << addr << "]\t" << insAddr
          << ": " << insDis << std::endl;
        if (!REG_valid(reg_r) || !checkAlreadyRegTainted(reg_r))
          removeMemTainted(addr);
        return ;
      }
  }
  if (checkAlreadyRegTainted(reg_r)){
    std::cout << std::hex << "[WRITE in " << addr << "]\t" << insAddr
      << ": " << insDis << std::endl;
    addMemTainted(addr);
  }
}
```

For the **STORE** instruction is the same thing. If the destination location is tainted, we check if the register is tainted. If it is false, we need to free the location memory. Otherwise if the register is tainted, we taint the memory destination.

3.3 - Registers spread

First, we add this new callback in our instruction handler. If the current instruction has two oprands and if the first operand is a register, we call the spreadRegTaint function.

```
else if (INS_OperandCount(ins) > 1 && INS_OperandIsReg(ins, 0)){
  INS_InsertCall(
      ins, IPOINT_BEFORE, (AFUNPTR)spreadRegTaint,
      IARG_ADDRINT, INS_Address(ins),
      IARG_PTR, new string(INS_Disassemble(ins)),
      IARG_UINT32, INS_OperandCount(ins),
      IARG_UINT32, INS_RegR(ins, 0),
      IARG_UINT32, INS_RegW(ins, 0),
      IARG_END);
}
```

Then, is the same thing than the taint memory. We need to check if the source and destination register is tainted or not. If the source register is tainted we taint the destination register, otherwise we remove the destination register.

```
VOID spreadRegTaint(UINT64 insAddr, std::string insDis, UINT32 opCount, REG reg_r, REG reg_w)
{
  if (opCount != 2)
    return;

  if (REG_valid(reg_w)){
    if (checkAlreadyRegTainted(reg_w) && (!REG_valid(reg_r)
        || !checkAlreadyRegTainted(reg_r))){
      std::cout << "[SPREAD]\t\t" << insAddr << ": " << insDis << std::endl;
      std::cout << "\t\t\toutput: "<< REG_StringShort(reg_w) << " | input: "
        << (REG_valid(reg_r) ? REG_StringShort(reg_r) : "constant") << std::endl;
      removeRegTainted(reg_w);
    }
    else if (!checkAlreadyRegTainted(reg_w) && checkAlreadyRegTainted(reg_r)){
      std::cout << "[SPREAD]\t\t" << insAddr << ": " << insDis << std::endl;
      std::cout << "\t\t\toutput: " << REG_StringShort(reg_w) << " | input: "
        << REG_StringShort(reg_r) << std::endl;
      taintReg(reg_w);
    }
  }
}
```

3.4 - Output

Ok, we will try to spread the taint on the following code.

```
int foo2(char a, char b, char c)
{
  a = 1;
  b = 2;
  c = 3;
  return 0;
}

int foo(char *buf)
{
  char c, b, a;

  c = buf[0];
  b = c;
  a = buf[8];
  foo2(a, b, c);
  return true;
}

int main(int ac, char **av)
{
  int fd;
  char *buf;

  if (!(buf = malloc(32)))
    return -1;

  fd = open("./file.txt", O_RDONLY);
  read(fd, buf, 32), close(fd);
  foo(buf);
}
```

As you can see below, we have a cool first PoC which spreads the taint via mem/reg.

```
$ ../../../pin -t ./obj-intel64/Taint.so -- ./test
[TAINT]                  bytes tainted from 0xb5b010 to 0xb5b030 (via read)
[READ in b5b010]         400649: movzx eax, byte ptr [rax]
                         eax is now tainted
[WRITE in 7fffa185d9ff]  40064c: mov byte ptr [rbp-0x1], al
                         7fffa185d9ff is now tainted
[READ in 7fffa185d9ff]   40064f: movzx eax, byte ptr [rbp-0x1]
                         eax is already tainted
[WRITE in 7fffa185d9fe]  400653: mov byte ptr [rbp-0x2], al
                         7fffa185d9fe is now tainted
[READ in b5b018]         40065a: movzx eax, byte ptr [rax+0x8]
                         eax is already tainted
[WRITE in 7fffa185d9fd]  40065e: mov byte ptr [rbp-0x3], al
                         7fffa185d9fd is now tainted
[READ in 7fffa185d9ff]   400661: movsx edx, byte ptr [rbp-0x1]
                         edx is now tainted
[READ in 7fffa185d9fe]   400665: movsx ecx, byte ptr [rbp-0x2]
                         ecx is now tainted
[READ in 7fffa185d9fd]   400669: movsx eax, byte ptr [rbp-0x3]
                         eax is already tainted
[SPREAD]                 40066d: mov esi, ecx
                         output: esi | input: ecx
                         esi is now tainted
[SPREAD]                 40066f: mov edi, eax
                         output: edi | input: eax
                         edi is now tainted
[WRITE in 7fffa185d9c4]  40061c: mov byte ptr [rbp-0x14], dil
                         7fffa185d9c4 is now tainted
[WRITE in 7fffa185d9c0]  400620: mov byte ptr [rbp-0x18], cl
                         7fffa185d9c0 is now tainted
[WRITE in 7fffa185d9bc]  400623: mov byte ptr [rbp-0x1c], al
                         7fffa185d9bc is now tainted
[SPREAD]                 400632: mov eax, 0x0
                         output: eax | input: constant
                         eax is now freed
[SPREAD]                 7fcccf0b960d: mov edi, eax
                         output: edi | input: eax
                         edi is now freed
[SPREAD]                 7fcccf0cf7db: mov edx, 0x1
                         output: edx | input: constant
                         edx is now freed
[SPREAD]                 7fcccf0cf750: mov esi, ebx
                         output: esi | input: ebx
                         esi is now freed
[READ in 7fcccf438140]   7fcccf11027e: mov ecx, dword ptr [rbp+0xc0]
                         ecx is now freed
```

You can see the source code of this example 2 here : example 2 (./taint_example_2.cpp).

4 - Follow your data

Following your data can be very interesting and this is very straightforward to implement. To follow your data, you just need to check for each instruction if one of their operands is a memory or register tainted. If it is true, you just display the current instruction. With that, you can display all instructions like `cmp al, Imm` which does not spread the taint. Displaying

these information can be very interesting if you would like to implement a concolic execution - see my previous post (http://shell-storm.org/blog/Concolic-execution-taint-analysis-with-valgrind-and-constraints-path-solver-with-z3/) on the concolic execution).

4.1 - Requirement

First, we add a new callback function in our instruction handler. This callback is exactly the same as the spreadReg function. We chose to duplicate the code to keep a clearer source.

```
if (INS_OperandCount(ins) > 1 && INS_OperandIsReg(ins, 0)){
  INS_InsertCall(
      ins, IPOINT_BEFORE, (AFUNPTR)followData,
      IARG_ADDRINT, INS_Address(ins),
      IARG_PTR, new string(INS_Disassemble(ins)),
      IARG_UINT32, INS_RegR(ins, 0),
      IARG_END);
}
```

As you can see below, the function that follows the data check if the **READ** register is tainted. If this is the case, it just display the current instruction.

```
VOID followData(UINT64 insAddr, std::string insDis, REG reg)
{
  if (!REG_valid(reg))
    return;

  if (checkAlreadyRegTainted(reg)){
      std::cout << "[FOLLOW]\t\t" << insAddr << ": " << insDis << std::endl;
  }
}
```

4.2 - Output

For this test we will just compare each character with a constant.

```
int foo(char *buf)
{
  if (buf[0] != 'A')
    return false;
  if (buf[1] != 'B')
    return false;
  if (buf[2] != 'C')
    return false;
  if (buf[3] != 'D')
    return false;
  return true;
}
```

As you can see below, the **cmp** instruction is displayed because **eax** is tainted.

```
$ ../../../pin -t ./obj-intel64/Taint.so -- ./test
[TAINT]             bytes tainted from 0x1ea4010 to 0x1ea4030 (via read)
[READ in 1ea4010]   400620: movzx eax, byte ptr [rax]
                    eax is now tainted
[FOLLOW]            400623: cmp al, 0x41
[READ in 1ea4011]   400636: movzx eax, byte ptr [rax]
                    eax is already tainted
[FOLLOW]            400639: cmp al, 0x42
[READ in 1ea4012]   40064c: movzx eax, byte ptr [rax]
                    eax is already tainted
[FOLLOW]            40064f: cmp al, 0x43
[READ in 1ea4013]   400662: movzx eax, byte ptr [rax]
                    eax is already tainted
[FOLLOW]            400665: cmp al, 0x44
[SPREAD]            400670: mov eax, 0x0
                    output: eax | input: constant
                    eax is now freed
```

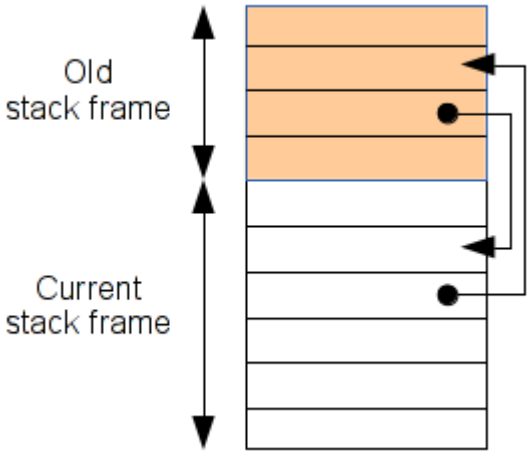
You can see the source code of this example 3 here : example 3 (./taint_example_3.cpp).

5 - Detect a use after free vulnerability

The taint analysis is cool but is mainly used to determine if it is a vulnerability can be triggered by the user. In this chapiter and the next, we will see how we can detect some vulnerabilities with a pattern matching and how it is very straightforward to combine these analysis.

5.1 - Via obsolete stack frame

First we will try to detect if we can find an access in **LOAD** or **STORE** to an obsolete stack frame, and if it can be controlled by the user. I know this bug is not a serious vector attack, but this blog post aims to show you what it is possible to do with Pin. So, imagine a memory access like that :



To test the obsolete stack frame detector, we will try our Pin tool on the following code.

```
char *ptr1;
char *ptr2;

char *foo2(char *buf)
{
  char d = buf[0]; /* 'd' is tainted because buf is controlled by the user */
  char e = buf[1]; /* 'e' is tainted because buf is controlled by the user */
  char f = 1;      /* 'f' is not tainted */

  ptr1 = &e;
  ptr2 = &f;

  return &d;
}

int foo1(char *buf)
{
  char a = *foo2(buf); /* UAF match          */
  char b = *ptr1;      /* UAF match          */
  char c = *ptr2;      /* UAF does not match */
}
```

As you can see above, the **foo2** function initializes the **ptr1** and **ptr2**, but they point on foo2's stack-frame. The function returns a pointer of its stack-frame. In this case, we can control only the '**a**' and '**b**' variables, because '**c**' is assigned with a constant and cannot be controlled by the user.

Below is the disassembly of **foo1** and **foo2**:

```
0000000000400614 <foo2>:
  400614:   55                      push   rbp
  400615:   48 89 e5                mov    rbp,rsp
  400618:   48 89 7d e8             mov    QWORD PTR [rbp-0x18],rdi
  40061c:   48 8b 45 e8             mov    rax,QWORD PTR [rbp-0x18]
  400620:   0f b6 00                movzx  eax,BYTE PTR [rax]
  400623:   88 45 ff                mov    BYTE PTR [rbp-0x1],al
  400626:   48 8b 45 e8             mov    rax,QWORD PTR [rbp-0x18]
  40062a:   0f b6 40 01             movzx  eax,BYTE PTR [rax+0x1]
  40062e:   88 45 fe                mov    BYTE PTR [rbp-0x2],al
  400631:   c6 45 fd 01             mov    BYTE PTR [rbp-0x3],0x1
  400635:   48 8d 45 fe             lea    rax,[rbp-0x2]
  400639:   48 89 05 20 0a 20 00    mov    QWORD PTR [rip+0x200a20],rax # 601060 <ptr1>
  400640:   48 8d 45 fd             lea    rax,[rbp-0x3]
  400644:   48 89 05 1d 0a 20 00    mov    QWORD PTR [rip+0x200a1d],rax # 601068 <ptr2>
  40064b:   48 8d 45 ff             lea    rax,[rbp-0x1]
  40064f:   5d                      pop    rbp
  400650:   c3                      ret

0000000000400651 <foo1>:
  400651:   55                      push   rbp
  400652:   48 89 e5                mov    rbp,rsp
  400655:   48 83 ec 18             sub    rsp,0x18
  400659:   48 89 7d e8             mov    QWORD PTR [rbp-0x18],rdi
  40065d:   48 8b 45 e8             mov    rax,QWORD PTR [rbp-0x18]
  400661:   48 89 c7                mov    rdi,rax
  400664:   e8 ab ff ff ff          call   400614 <foo2>
  400669:   0f b6 00                movzx  eax,BYTE PTR [rax]
  40066c:   88 45 ff                mov    BYTE PTR [rbp-0x1],al
  40066f:   48 8b 05 ea 09 20 00    mov    rax,QWORD PTR [rip+0x2009ea] # 601060 <ptr1>
  400676:   0f b6 00                movzx  eax,BYTE PTR [rax]
  400679:   88 45 fe                mov    BYTE PTR [rbp-0x2],al
  40067c:   48 8b 05 e5 09 20 00    mov    rax,QWORD PTR [rip+0x2009e5] # 601068 <ptr2>
  400683:   0f b6 00                movzx  eax,BYTE PTR [rax]
  400686:   88 45 fd                mov    BYTE PTR [rbp-0x3],al
  400689:   c9                      leave
  40068a:   c3
```

To detect this issue, you can just check the current stack pointer with the source pointer. If the current stack pointer is above than the source location it means that we have a potential obsolete stack-frame use.

```
if (sp > addr && addr > 0x700000000000)
  std::cout << std::hex << "[UAF in " << addr << "]\t" << insAddr \
    << ": " << insDis << std::endl;
```

Here is the output displaying the taint, and the obsolete stack-frame usage:

```
$ ../../../pin -t ./obj-intel64/Taint.so -- ./test
[TAINT]                   bytes tainted from 0x611010 to 0x611030 (via read)
[READ in 611010]          400620: movzx eax, byte ptr [rax]
                          eax is now tainted
[WRITE in 7fffd14495e7]   400623: mov byte ptr [rbp-0x1], al
                          7fffd14495e7 is now tainted
[READ in 611011]          40062a: movzx eax, byte ptr [rax+0x1]
                          eax is already tainted
[WRITE in 7fffd14495e6]   40062e: mov byte ptr [rbp-0x2], al
                          7fffd14495e6 is now tainted
[READ in 7fffd14495e7]    400669: movzx eax, byte ptr [rax]
                          eax is already tainted
[UAF in 7fffd14495e7]     400669: movzx eax, byte ptr [rax]
[WRITE in 7fffd144960f]   40066c: mov byte ptr [rbp-0x1], al
                          7fffd144960f is now tainted
[READ in 7fffd14495e6]    400676: movzx eax, byte ptr [rax]
                          eax is already tainted
[UAF in 7fffd14495e6]     400676: movzx eax, byte ptr [rax]
[WRITE in 7fffd144960e]   400679: mov byte ptr [rbp-0x2], al
                          7fffd144960e is now tainted
[READ in 7fffd14495e5]    400683: movzx eax, byte ptr [rax]
                          eax is now freed
```

This is the same thing if you write in the obsolete stack frame. In the following code we write in an obsolete stack frame.

```
char *foo2(char *buf)
{
  char d = buf[0];
  return &d;
}

int foo1(char *buf)
{
  *foo2(buf) = 1; /* UAF match */
}
```

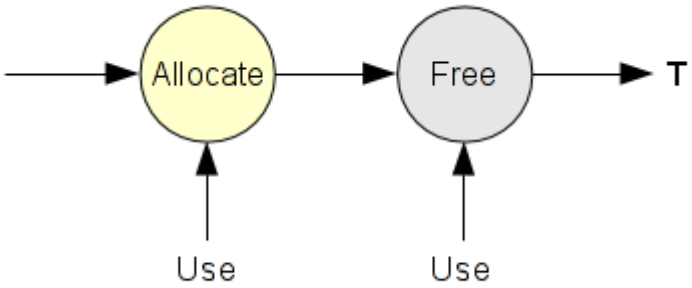
The Pin tool tells us at the address `400644` we have an arbitrary write in an obsolete stack-frame.

```
$ ../../../pin -t ./obj-intel64/Taint.so -- ./test
[TAINT]                   bytes tainted from 0x19fd010 to 0x19fd030 (via read)
[READ in 19fd010]         400620: movzx eax, byte ptr [rax]
                          eax is now tainted
[WRITE in 7fffd3fd0f37]   400623: mov byte ptr [rbp-0x1], al
                          7fffd3fd0f37 is now tainted
[WRITE in 7fffd3fd0f37]   400644: mov byte ptr [rax], 0x1
                          7fffd3fd0f37 is now freed
[UAF in 7fffd3fd0f37]     400644: mov byte ptr [rax], 0x1
```

You can see the source code of this example 4 here : example 4 (./taint_example_4.cpp).

5.2 - Classical use after free

An use after free bug occurs when we continue to use a pointer after it has been freed.

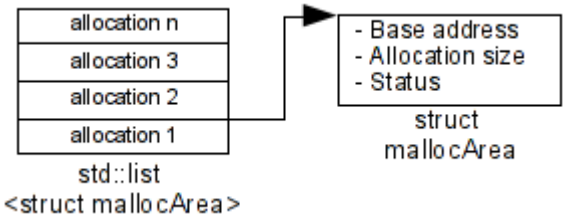


This error is widely known in the C++ world.. Imagine an object with a classical public/private methods/var. The first object '**A**' is initialized (We do not control this object), then some time later this object is freed. After we have a second same object '**B**' (We can control it), the allocation will be potentially at the same memory location. If in this object you can control these private/public data, when the object '**A**' is used, this is the object '**B**' which will be targeted.

In this little chapiter, we will try to detect these issues.

5.2.1 - Methodology

For this PoC I have chosen to catch all calls at the **malloc** and **free** function. With Pin it is very straightforward to catch these calls via their symbols. When a **malloc** occurs, we save in a list these information, like the base address, the size and we assigns a status (**ALLOCATE** or **FREE**). When a **free** occurs, we set the flag to **FREE**.



Then, when we have a **LOAD** or **STORE** operation, we check if this address is in our list and we check its flag. If the flag is **FREE** and that we have an accesss in **LOAD** or **STORE**, that means we have a potential use after free bug.

5.2.2 - Pin API - Symbols

With Pin it is possible to execute a callback when a symbol is triggered. Here, we add a callback when the symbol **malloc** or **free** occurs. First, we need to initialize the Pin symbols.

```
PIN_InitSymbols();
if(PIN_Init(argc, argv)){
    return Usage();
}
```

Then, in the handler Image we can add a callback for a specific symbols.

```
VOID Image(IMG img, VOID *v)
{
  RTN mallocRtn = RTN_FindByName(img, "malloc");
  RTN freeRtn = RTN_FindByName(img, "free");

  if (RTN_Valid(mallocRtn)){
    RTN_Open(mallocRtn);

    RTN_InsertCall(
        mallocRtn,
        IPOINT_BEFORE, (AFUNPTR)callbackBeforeMalloc,
        IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
        IARG_END);

    RTN_InsertCall(
        mallocRtn,
        IPOINT_AFTER, (AFUNPTR)callbackAfterMalloc,
        IARG_FUNCRET_EXITPOINT_VALUE,
        IARG_END);

    RTN_Close(mallocRtn);
  }

  if (RTN_Valid(freeRtn)){
    RTN_Open(freeRtn);
    RTN_InsertCall(
        freeRtn,
        IPOINT_BEFORE, (AFUNPTR)callbackBeforeFree,
        IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
        IARG_END);
    RTN_Close(freeRtn);
  }
}

int main(int argc, char *argv[])
{
    ...
    IMG_AddInstrumentFunction(Image, 0);
    ...
    return 0;
}
```

In these callbacks, we save and monitor all allocations. Then, in our access memory callback (**LOAD**/**STORE**), we check if the destination/source address is allocated or freed.

```
for(i2 = mallocAreaList.begin(); i2 != mallocAreaList.end(); i2++){
  if (addr >= i2->base && addr < (i2->base + i2->size) && i2->status == FREE){
    std::cout << std::hex << "[UAF in " << addr << "]\t" << insAddr << ": "
      << insDis << std::endl;
    return;
  }
}
```

5.2.3 - Test on a C based program

First test in a C-based program. Here, we allocate a memory area of 32 bytes. The first and third **STORE** are OK, because **buf** is allocated but when the second **STORE** occurs **buf** is freed.

```
int main(int ac, char **av)
{
  char *buf;
  char c;

  if (!(buf = malloc(32)))
    return -1;

  c = buf[0];           /* UAF not match */
  free(buf);
  c = buf[0];           /* UAF match     */
  buf = malloc(32);
  c = buf[0];           /* UAF not match */
}
```

Our output looks like that.

```
$ ../../../pin -t ./obj-intel64/Taint.so -- ./test
[INFO]           malloc(32) = 618010
[INFO]           free(618010)
[UAF in 618010]  7f0257cb9ccb: mov qword ptr [rbx+0x10], rdx
[UAF in 618010]  4005c9: movzx eax, byte ptr [rax]
[UAF in 618010]  7f0257cba77f: mov r8, qword ptr [r14+0x10]
[INFO]           malloc(32) = 618010
$
```

5.2.4 - Test based on C++

Second test in C++ based program.

```
class Test
{
  private:
    int a;
    void foo(void);

  public:
    Test(int num) { this->a = num; };
    ~Test() {};
    void wrapper(void);
};

void Test::foo(void) {
  std::cout << this->a << std::endl;
}

void Test::wrapper(void) {
  this->foo();
}

int main()
{
  Test *ptr = new Test(1234);
  Test *old = ptr;

  ptr->wrapper();
  delete ptr;
  ptr->wrapper();
  ptr = new Test(4321);
  old->wrapper();
}
```

Our output looks like that.

```
$ ../../../pin -t ./obj-intel64/Taint.so -- ./testcpp
[INFO]              malloc(4) = 1c04010
1234
[INFO]              free(1c04010)
[UAF in 1c04010]    7f0913b5accb: mov qword ptr [rbx+0x10], rdx
0
[UAF in 1c04010]    7f0913b5b77f: mov r8, qword ptr [r14+0x10]
[INFO]              malloc(4) = 1c04010
4321
```

You can see the source code of this example 5 here : example 5 (./taint_example_5.cpp).

6 - Detect pointer utilization without check

During the previous chapter, I thought that it was fun to detect the pointer utilization without check. In userland is not really useful, except for crashing the program. But this paper is just to know what it is possible to do with Pin, and it may give people other ideas.

So, imagine the following code :

```
int main(int ac, char **av)
{
  char c, *buf;

  buf = malloc(32);
  c = buf[0];
}
```

As you know, if **malloc** fails, it returns **NULL** and if the source location is **NULL** and when the **STORE** occurs, we get a page fault. To resolve that, we just need to check if the pointer is **NULL**, like this :

```
int main(int ac, char **av)
{
  char c, *buf;

  if(!(buf = malloc(32)))
    return -ENOMEM;
  c = buf[0];
}
```

6.1 - Via pattern matching

For the pattern matching we need to analyze the **ASM** representation of this check on different forms.

```
/* Without optimization */

if(!(buf = malloc(32)))       call   400440 <malloc@plt>
  ...                         mov    QWORD PTR [rbp-0x8],rax
                              cmp    QWORD PTR [rbp-0x8],0x0

-------------------------------------------------------------

/* With optimization -O3 */

if(!(buf = malloc(32)))       call   400440 <malloc@plt>
  ...                         test   rax,rax

-------------------------------------------------------------

/* Without optimization */

buf = malloc(32);             call   400440 <malloc@plt>
if(buf == NULL)               mov    QWORD PTR [rbp-0x8],rax
   ...                        cmp    QWORD PTR [rbp-0x8],0x0


-------------------------------------------------------------

/* Check in other stack frame without optimization */

buf = malloc(32);             call   400480 <malloc@plt>
chk(buf);                     mov    QWORD PTR [rbp-0x8],rax
                              mov    rax,QWORD PTR [rbp-0x8]
                              mov    rdi,rax
                              call   400584 <chk>

void chk(char *buf){          mov    QWORD PTR [rbp-0x8],rdi
  if (!buf)                   cmp    QWORD PTR [rbp-0x8],0x0
    ...
}
```

As you can see above, if you compile without optimization, we get a potential '**cmp mem, imm**' instruction. Otherwise you get a '**test reg, reg**' instruction. But it is not reliable because that depends of your compiler. So, in our case we add two new callbacks for the **CMP** and **TEST** instruction.

```
else if (INS_Opcode(ins) == XED_ICLASS_CMP && INS_OperandIsMemory(ins, 0)){
  INS_InsertCall(
      ins, IPOINT_BEFORE, (AFUNPTR)cmpInst,
      IARG_ADDRINT, INS_Address(ins),
      IARG_PTR, new string(INS_Disassemble(ins)),
      IARG_MEMORYOP_EA, 0,
      IARG_END);
}
else if (INS_Opcode(ins) == XED_ICLASS_TEST && INS_OperandCount(ins) >= 2 &&
         REG_valid(INS_OperandReg(ins, 0)) && REG_valid(INS_OperandReg(ins, 1))){
  INS_InsertCall(
      ins, IPOINT_BEFORE, (AFUNPTR)testInst,
      IARG_ADDRINT, INS_Address(ins),
      IARG_PTR, new string(INS_Disassemble(ins)),
      IARG_REG_VALUE, INS_OperandReg(ins, 0),
      IARG_REG_VALUE, INS_OperandReg(ins, 1),
      IARG_END);
}
```

And in our callbacks we just see if the **TEST** or **CMP** instruction checks a allocated pointer. If it is **True** we update the **check flag** to **CHECKED**.

```
VOID cmpInst(UINT64 insAddr, std::string insDis, UINT64 memOp)
{
  list<struct mallocArea>::iterator i;
  UINT64 addr = memOp;

  for(i = mallocAreaList.begin(); i != mallocAreaList.end(); i++){
    if (*(UINT64 *)addr == i->base){
      std::cout << std::hex << "[PTR " << *(UINT64 *)addr << " checked]\t\t\t"
        << insAddr << ": " << insDis << std::endl;
      i->check = CHECKED;
    }
  }
}

VOID testInst(UINT64 insAddr, std::string insDis, ADDRINT val_r0, ADDRINT val_r1)
{
  list<struct mallocArea>::iterator i;

  for(i = mallocAreaList.begin(); i != mallocAreaList.end(); i++){
    if (val_r0 == val_r1 && val_r0 == i->base){
      std::cout << std::hex << "[PTR " << val_r0 << " checked]\t\t\t" << insAddr
        << ": " << insDis << std::endl;
      i->check = CHECKED;
    }
  }
}
```

Then, when we got a **LOAD** or **STORE** instruction we just need to see the **check flag**. If the flag is **!CHECKED**, that means the pointer has potentially been not checked.

```
if (i->base == addr && i->check != CHECKED)
   std::cout << std::hex << "[WRITE in " << addr << " without check]\t\t"
     << insAddr << ": " << insDis << std::endl;
```

6.2 - Output

We will compile the following program with **-O0** and **-O3**.

```
int main(int ac, char **av)
{
  char c, *buf;

  if (!(buf = malloc(32)))
    return -1;
  c = buf[0];
  printf("%x\n", c);
}
```

And we get something like that:

```
$ ../../../pin -t ./obj-intel64/Taint.so  -- ./test_without_opti
[INFO]                      malloc(32) = 1909010
[PTR 1909010 checked]       4005a1: cmp qword ptr [rbp-0x8], 0x0
0

$ ../../../pin -t ./obj-intel64/Taint.so  -- ./test_with_opti
[INFO]                      malloc(32) = e78010
[PTR e78010 checked]        4004ce: test rax, rax
0
```

Same thing but now without check.

```
buf = malloc(32);
c = buf[0];
printf("%x\n", c);
```

And we get something like that:

```
$ ../../../pin -t ./obj-intel64/Taint.so  -- ./test
[INFO]                              malloc(32) = 7ee010
[READ in 7ee010 without check]      4004ce: movsx edx, byte ptr [rax]
0
```

I have also tested it on a real binary like **/usr/bin/id** and I got this output (./output_id.txt). You can see the source code of this example 6 here: example 6 (./taint_example_6.cpp).

7 - Conclusion

7.1 - Taint analysis and pattern matching with Pin

This paper was just my personal notes, when last weeks I wanted to play with Pin. I wanted to know what is really possible to do with Pin. For my first approaches, the API is really easy to use and Pin has a great documentation. The negative point is that it lacks of an intermediate representation (IR) - It is just my personnal opinion, maybe it is not the Pin philosophy to give us an IR. In my case, when I wanted to do a taint analysis, without IR it is really boring. For example with an IR like VEX (Valgrind), you do not need to monitor the memory access and all is on the SSA form, which give us more flexibility to do the analysis. For the conclusions, the taint analysis with Pin is possible but boring; I think it is not really made for that... Then if want do this with Pin and that you want a perfect taint, we need to taint the bits and not the byte like in my examples and you must not lose one bit, otherwise your tree taint is corrupted. Then if you taint the

bits, your taint tree will consumes a lot of memory. Same for the pattern matching, without IR it is possible but really boring, complicated and not reliable. To conclude, I think Pin is cool just for small examples, or quick analysis ; but not for a serious project.

7.2 - References

7.2.1 - Web references

- Pin tool (http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool)
- Pin API Reference (http://software.intel.com/sites/landingpage/pintool/docs/55942/Pin/html/group__API__REF.html)
- Pin User Guide (http://software.intel.com/sites/landingpage/pintool/docs/58423/Pin/html/)

7.2.2 - My Pin tool sources

- Example 1 - Simple taint a memory area (./taint_example_1.cpp)
- Example 2 - Spread the taint in memory and registers (./taint_example_2.cpp)
- Example 3 - Spread the taint and follow your data (./taint_example_3.cpp)
- Example 4 - Obsolete stack frame access detection (./taint_example_4.cpp)
- Example 5 - Classical Use after free pattern matching (./taint_example_5.cpp)
- Example 6 - Pointer without check detection (./taint_example_6.cpp)

7.3 - Special thanks

I would like to thank those guys which gave me a lot of feedbacks **:-)**

- Axel "0vercl0k" Souchet (https://twitter.com/0vercl0k)
- Kevin Szkudlapski (https://twitter.com/medusa_disasm)