Taint Analysis

Contents

- Pin Tool
  - Introduction
  - Intel PIN Capability
  - How to instrumentation
  - How to Pass Parameters
  - Instrumentation granularity
- Dynamic Taint Analysis
  - Classify of taint analysis
  - Basic Concept
  - Introduction
  - Byte or bit
  - Shadow Memory
  - Dynamic Taint Analysis

Pin tools

Instrumentation

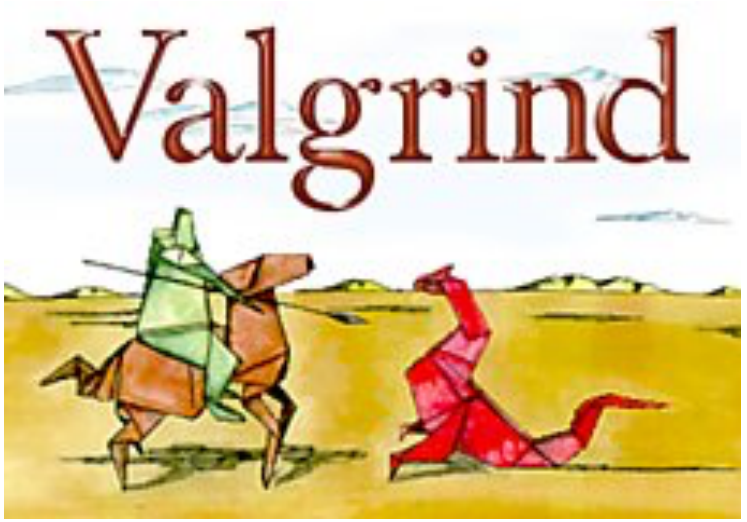
- A technique that inserts code into a program to collect run-time information or change its behavior

Different Instrumentations

- Source-Code Instrumentation
  - Compiler Plugin
    - ✓ Insert code where compile the source to binary
    - ✓ High efficient
- Static Binary Instrumentation
  - Binary rewriter
    - ✓ Disassembling and recompile
    - ✓ Difficult to ensure correctness
- Dynamic Binary Instrumentation
  - Dynamic Binary Instrumentation Tool
    - ✓ Instrument code just before it runs
    - ✓ No need to recompile or re-link
    - ✓ Analyze and modify code at runtime

Dynamic Binary Instrumentation

- Intel PIN
- Valgrind
- QEMU

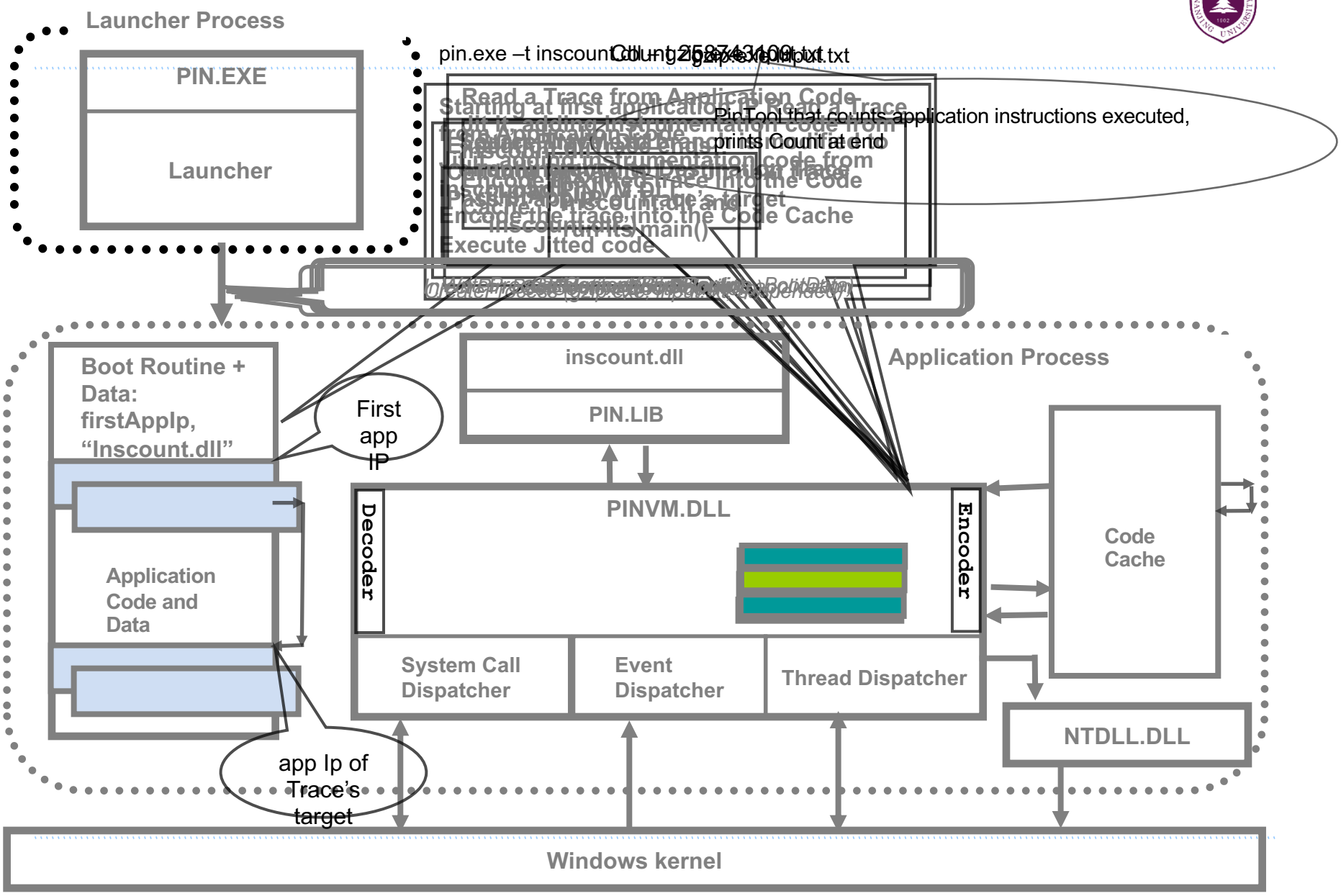


Intel Pin Capability

- Binary Analysis:
  - Trace Control Flow or Data Flow
  - Hook function, signals and system call
  - Multi-Thread support

- Change program behavior:
  - Add/delete instructions/basic blocks/functions
  - Change register values
  - Change control flow
  - Change memory values

Pin Work Flow Demonstration



How to instrumentation

Insert callback function for instructions, basic blocks, functions and image.

e.g., Instruction Instrumentation

```
int main(int argc, char * argv[])
{
    // Initialize pin
    if (PIN_Init(argc, argv)) return Usage();

    OutFile.open(KnobOutputFile.Value().c_str());

    // Register Instruction to be called to instrument instructions
    INS_AddInstrumentFunction(Instruction, 0);

    // Register Fini to be called when the application exits
    PIN_AddFiniFunction(Fini, 0);

    // Start the program, never returns
    PIN_StartProgram();

    return 0;
}
```

How to instrumentation

```
// Pin calls this function every time a new instruction is encountered
VOID Instruction(INS ins, VOID *v)
{
    // Insert a call to docount before every instruction, no arguments are passed
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_END);
}
```

How to Pass Parameters

```
INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)ifun,
IARG_TYPE, IARG, …… IARG_END);
```

IARG_TYPE:

- IARG_ADDRINT
- IARG_PTR
- IARG_BOOL
- IARG_UINT32
- IARG_UINT64
- IARG_INST_PTR
- IARG_REG_VALUE
- IARG_REG_REFERENCE
- IARG_REG_CONST_REFERENCE
- ……

IARG:

- INS_Address(ins)
- INS_OperandReg(ins, 0)
- INS_MemoryOperandCount(ins)
- INS_Valid(ins)
- ……

Instrumentation Granularity:

- Instruction instrumentation
- Basic block instrumentation
  - A sequence of instructions terminated at a control-flow changing instruction
  - Single entry, single exit
- Trace instrumentation
  - A sequence of basic blocks terminated at an unconditional control-flow changing instruction
  - Single entry, multiple exits
- Routine instrumentation
- Image instrumentation

APIs:

- INS_AddInstrumentFunction (INSCALLBACK fun, VOID *val)
- TRACE_AddInstrumentFunction (TRACECALLBACK fun, VOID *val)
- RTN_AddInstrumentFunction (RTNCALLBACK fun, VOID *val)
- IMG_AddInstrumentFunction (IMGCALLBACK fun, VOID *val)
- PIN_AddFiniFunction (FINICALLBACK fun, VOID *val)
- PIN_AddDetachFunction (DETACHCALLBACK fun, VOID *val)

Compare with Trace and Basic Block

```
sub     $0xff, %edx
cmp     %esi, %edx
jle     <L1>
```

```
mov     $0x1, %edi
add     $0x10, %eax
jmp     <L2>
```

1 Trace, 2 BBs, 6 insts

Taint Analysis

- Classify of taint analysis
- Basic Concept
- Introduction
- Byte or bit
- Shadow Memory
- Dynamic Taint Analysis

Classify Of taint Analysis

- Static Taint Analysis
  - The advantage of using static analysis is the fact that it provides better code coverage than dynamic analysis.
  - On the other hand, the principal disadvantage of the static analysis is that it's not as accurate than the dynamic analysis - It cannot access the runtime information for example. We can't retrieve registers or memory values.

- Dynamic Taint Analysis
  - Dynamic analysis we can't cover all the code but you will be more reliable.

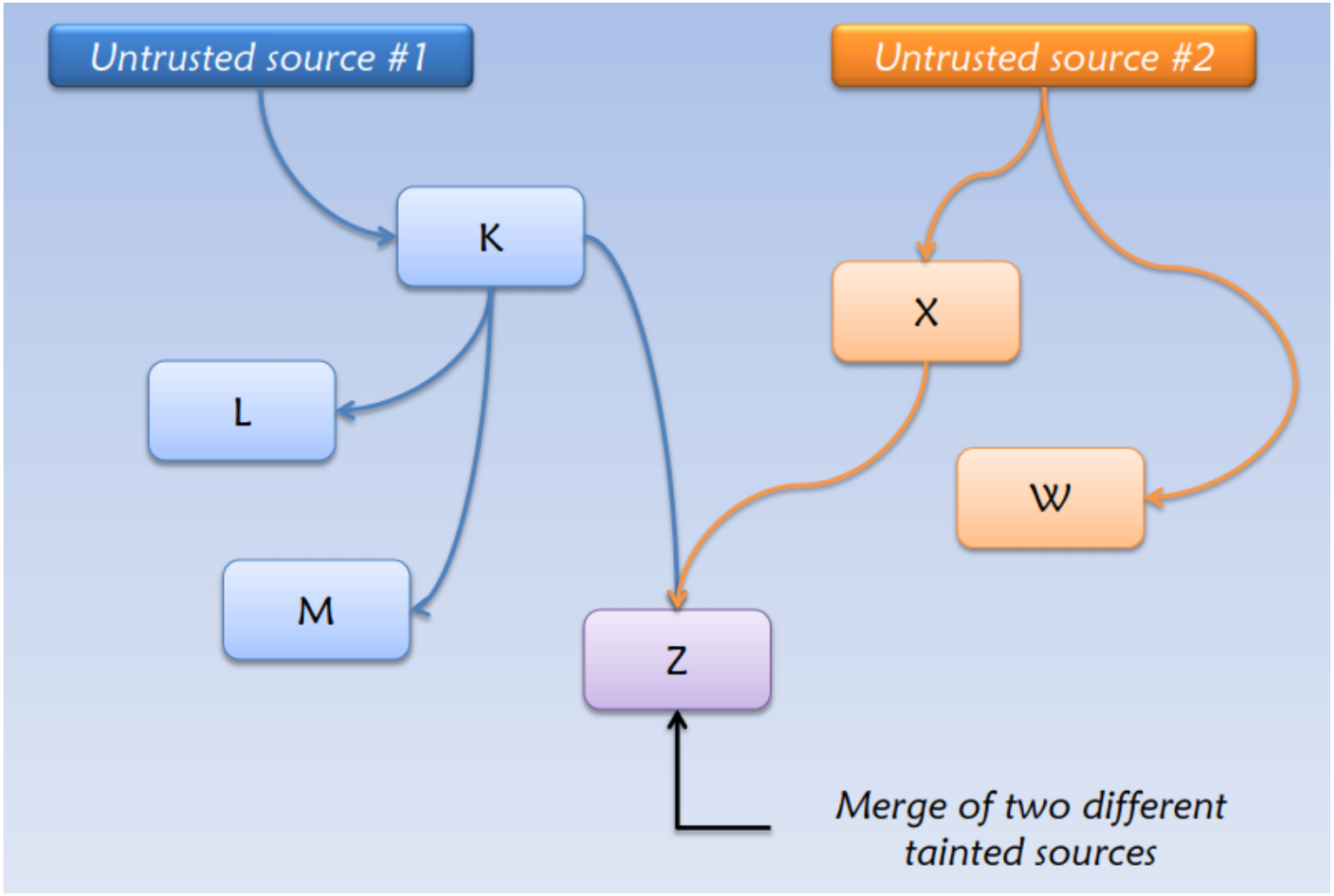
Basic concept

Taint propagation:

- Taint If an operation uses the value of some tainted object, say X, to derive a value for another, say Y, then object Y becomes tainted. Object X tainted the object Y

- Taint operator ***t***
- ***X → t(Y)***
- Taint operator is transitive
  - X → t(Y) and Y → t(Z), then X → t(Z)

Taint propagation



Basic concept

- Taint Sources: program, or memory locations, where data of interest enter the system and subsequently get tagged. For the convenience of description, we use the user input as the taint source in this course.
- Taint Tracking: process of propagating data tags according to program semantics
- Taint Sinks: program, or memory locations, where checks for tagged data can be made

Introduction

Taint analysis is used to know at a program point what part of memory or register are controllable by the some data we are interested, for example:user input.

According to the instruction semantics the taint is spread over the execution.

Introduction

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);
}
```

In the example 1, 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.

Introduction

```
/* 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 example 2, 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.

Byte or bit ?

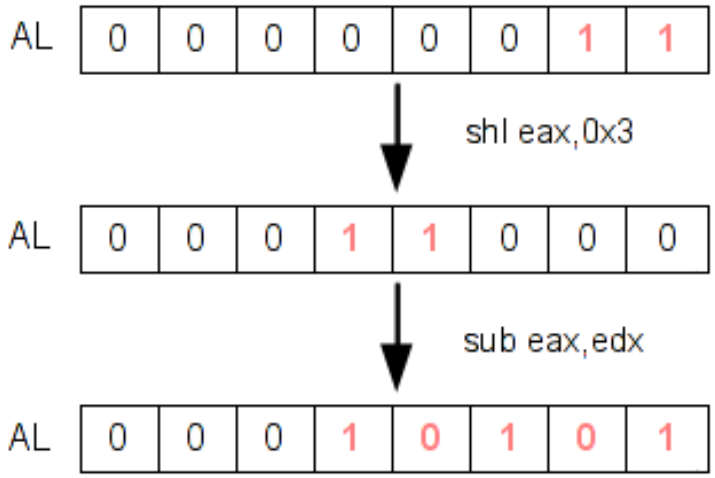
One of these problems is to determine what method is the more accurate to do a taint with a great precision. For example, what are we supposed to do when a controlled byte is multiplied and stored somewhere in memory ? Should we taint the destination variable ? See the following code.

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

```
call    atoi@plt
mov     eax, edx
cmp     eax, $0
jse     next
cmp     eax, $4
jne     next
shl     eax, 0x3
sub     eax,  edx
mov     eax, DWORD PTR[rbp-0x4]
next:
mov     DWORD PTR[rbp-0x4], eax
leave
ret
```

Byte or bit ?

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.



Byte or bit ?

Byte taint analysis assert b is tainted.

Bit taint analysis assert b is not tainted.

```
1     char a, b, w1, w2
2     char odd_bits = 01010101b
3     char even_bits = 10101010b
4     a = read()
5     b = 10
6     w1 = (a ∧ even_bits)
           ∨ (b ∧ odd_bits)
7     w2 = (a ∧ odd_bits)
           ∨ (b ∧ even_bits)
           ...
i     a = (w1 ∧ even_bits)
           ∨ (w2 ∧ odd_bits)
i+1   b = (w1 ∧ odd_bits)
           ∨ (w2 ∧ even_bits)
i+2   print(b)
```

Byte or bit ?

So, what to do? Tainting bytes is easier and light or tainting bits controlled by the user?

If you taint bytes, it will be easier but not reliable.

If we taint bits, it will be harder and more difficult to manage the taint tree but it will be 99% reliable.

Taint bytes is enough for most situation.

Dynamic Taint Analysis

How to do the dynamic taint analysis?

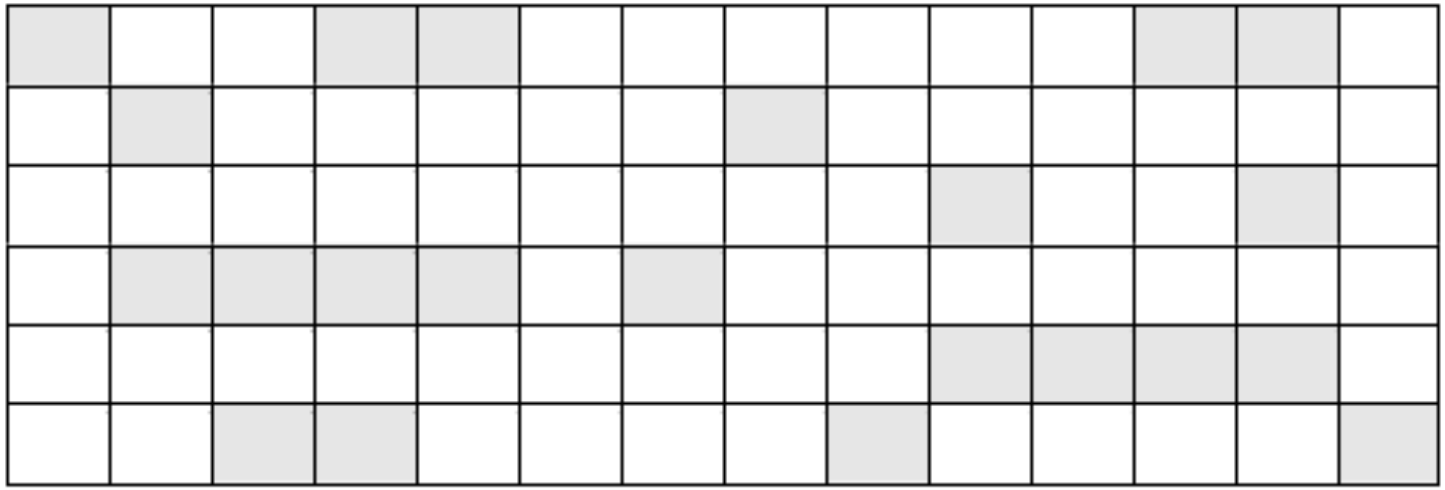
Dynamic Taint Analysis

In order to do this, we need a dynamic binary instrumentation(DBI) 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).

We choose to use Pin: a C++ dynamic binary instrumentation framework (without IR) written by Intel.

Shadow Memory



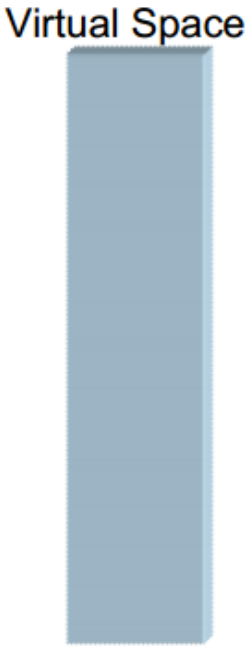
We user shadow memory to mark all address can be tainted by originate data we interested.

Shadow Memory

- Shadow Memory: Shadow memory describes a computer science technique in which potentially every byte used by a program during its execution has a shadow byte or bytes.
- These shadow bytes are typically invisible to the original program and are used to record information about the original piece of data.

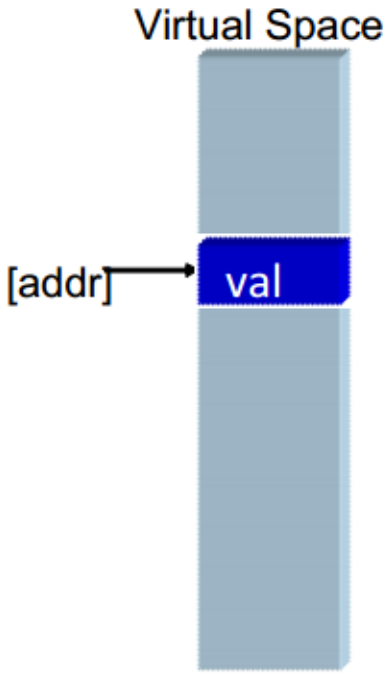
Shadow Memory

- Shadow Memory
  - We need a mapping
    - ✓ Addr → Abstract State
    - ✓ Register → Abstract



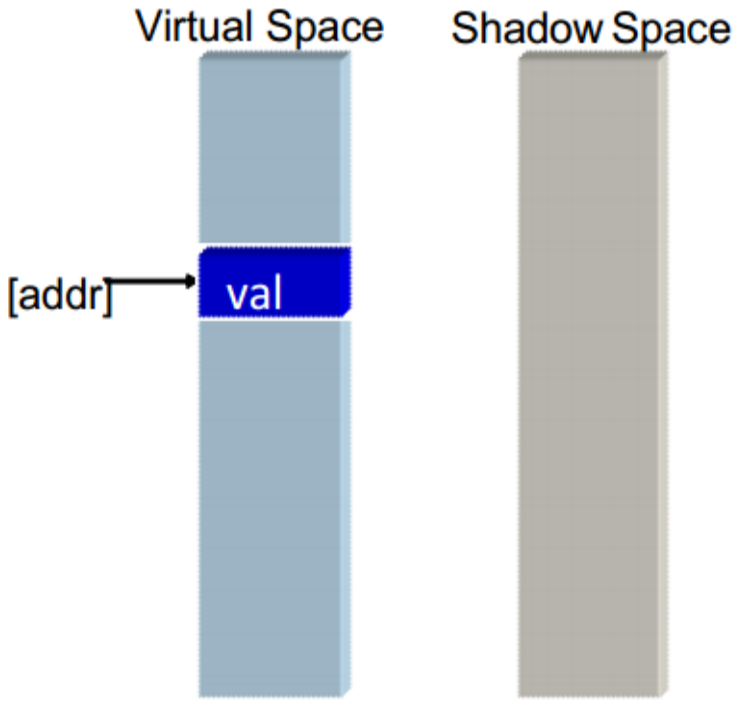
Shadow Memory

- Shadow Memory
  - We need a mapping
    - ✓ Addr → Abstract State
    - ✓ Register → Abstract



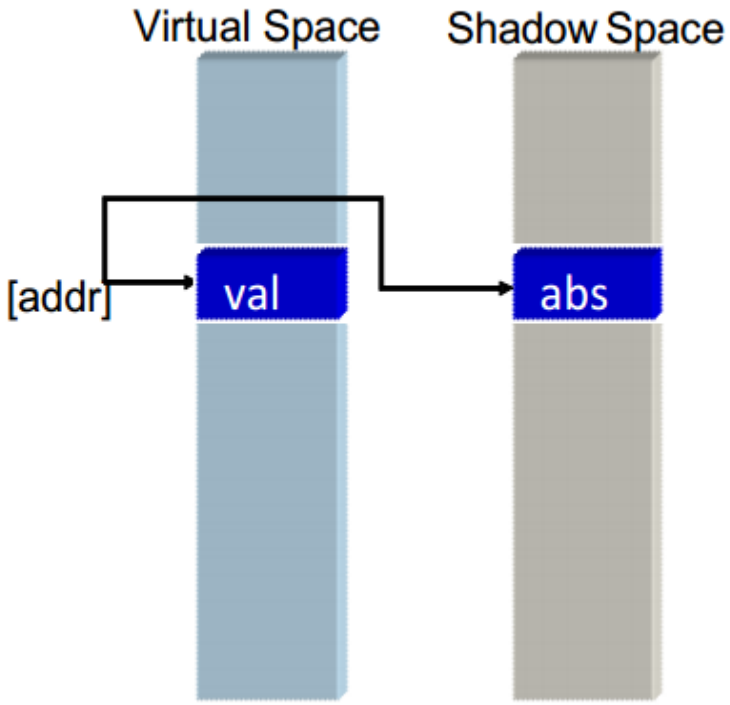
Shadow Memory

- Shadow Memory
  - We need a mapping
    - ✓ Addr → Abstract State
    - ✓ Register → Abstract



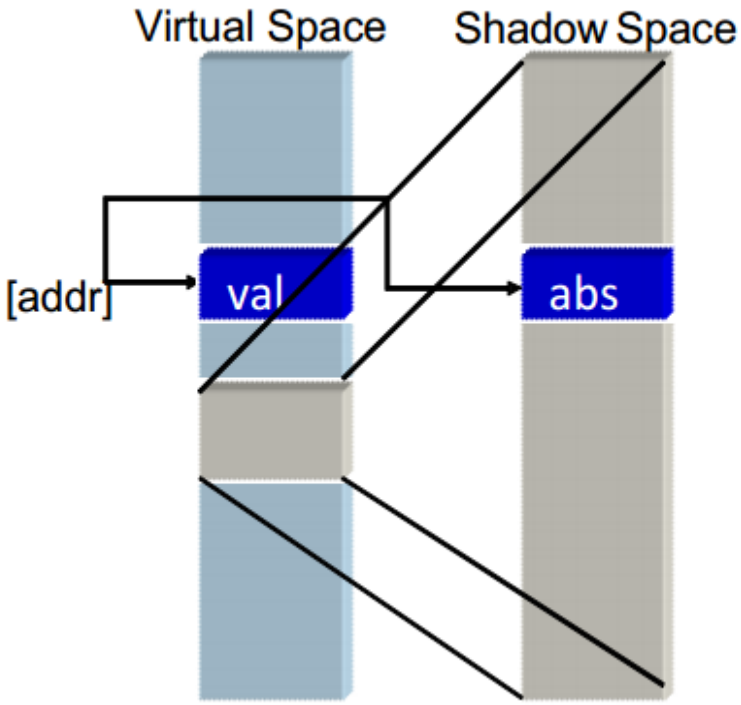
Shadow Memory

- Shadow Memory
  - We need a mapping
    - ✓ Addr → Abstract State
    - ✓ Register → Abstract



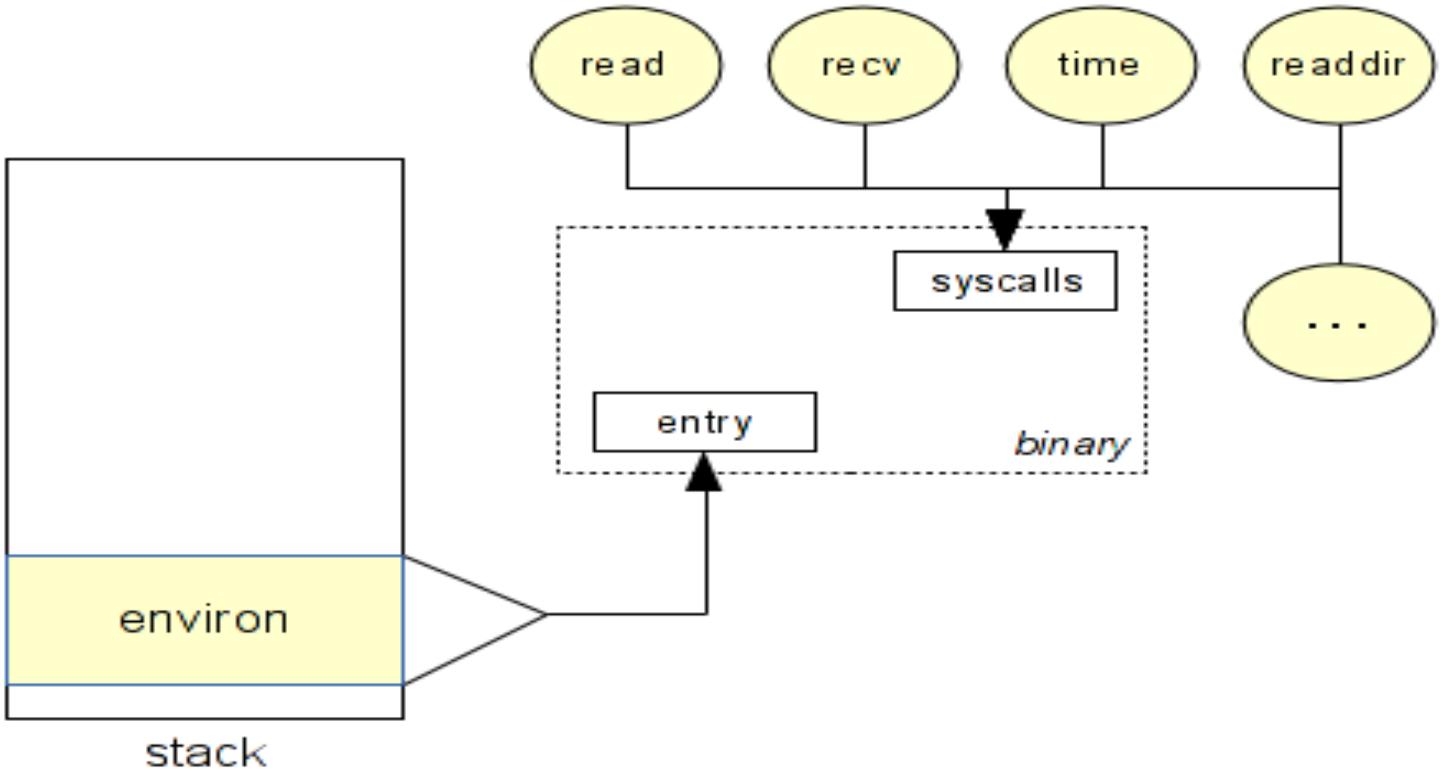
Shadow Memory

- Shadow Memory
  - We need a mapping
    - ✓ Addr → Abstract State
    - ✓ Register → Abstract



Dynamic taint Analysis

Firstly 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.



Dynamic Taint Analysis

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

Catch the syscalls

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.

The second argument is the start of memory address which the syscall is writing to.

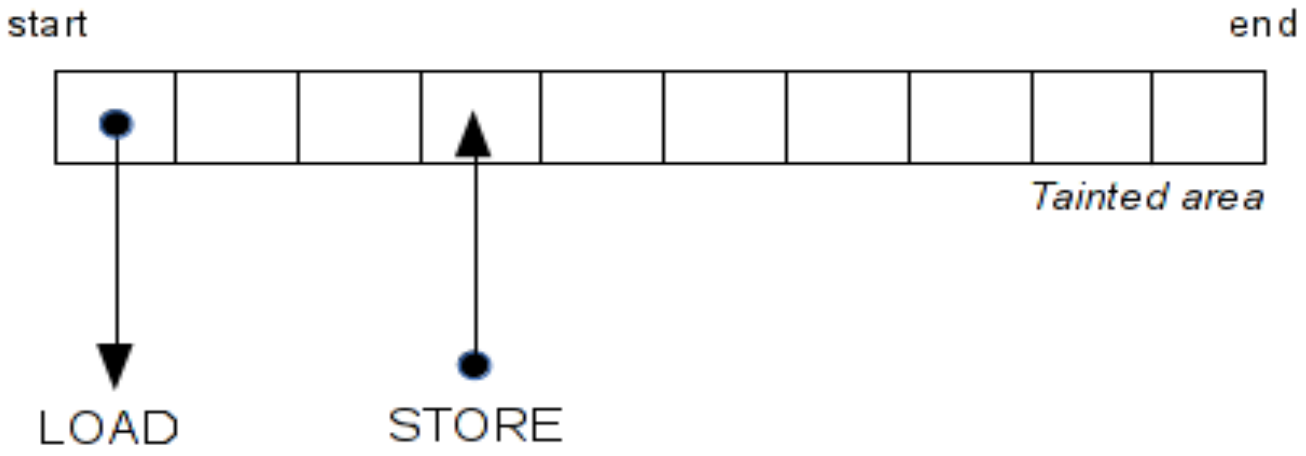
The third argument is the length of data to write to the memeory.

Catch the syscalls

```
VOID Syscall_entry(THREADID thread_id, CONTEXT *ctx, SYSCALL_STANDARD std, void *v)
{
  unsigned int i;
  UINT32 start, size;
  if (PIN_GetSyscallNumber(ctx, std) == __NR_read){
      TRICKS(); /* tricks to ignore the first open */
      start = static_cast<UINT64>((PIN_GetSyscallArgument(ctx, std, 1)));
      size  = static_cast<UINT64>((PIN_GetSyscallArgument(ctx, std, 2)));
      for (i = 0; i < size; i++)
        addressTainted.push_back(start+i);
      std::cout << "[TAINT]\t\t\tbytes tainted from " << std::hex << "0x" << start << " to 0x" <
  }
}
```

Catch the LOAD and STORE instructions

Now we need to catch 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.



Catch the LOAD and STORE instructions

```
if (INS_OperandCount(ins) > 1 && 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_UINT32, INS_OperandCount(ins),
      IARG_UINT32, INS_OperandReg(ins, 0),
      IARG_MEMORYOP_EA, 0,
      IARG_REG_VALUE, REG_STACK_PTR,
      IARG_END);
}
else if (INS_OperandCount(ins) > 1 && INS_MemoryOperandIsWritten(ins, 0)){
  INS_InsertCall(
      ins, IPOINT_BEFORE, (AFUNPTR)WriteMem,
      IARG_ADDRINT, INS_Address(ins),
      IARG_PTR, new string(INS_Disassemble(ins)),
      IARG_UINT32, INS_OperandCount(ins),
      IARG_UINT32, INS_OperandReg(ins, 1),
      IARG_UINT32, INS_OperandReg(ins, 0),
      IARG_MEMORYOP_EA, 0,
      IARG_REG_VALUE, REG_STACK_PTR,
      IARG_END);
}
```

Hook Load Instruction

```
VOID ReadMem(UINT32 insAddr, std::string insDis, UINT32 OperandCount,REG reg_r, UINT32 memOp, UINT32 sp)
{
  list<UINT32>::iterator i;
  UINT32 addr = memOp;
  if (OperandCount != 2)
    return;
  for(i = addressTainted.begin(); i != addressTainted.end(); i++){
      if (addr == *i){
        if(insAddr<=0x8049000)
                std::cout << std::hex <<insAddr<< ":\t"  <<"[READ in " << addr << "][T]" <<" insDis:"<<ir
        taintReg(reg_r);
        return ;
      }
  }
  /* if mem != tained and reg == taint => free the reg */
  if (checkAlreadyRegTainted(reg_r)){
        if(insAddr<=0x8049000)
        std::cout << std::hex <<insAddr<< ":\t"  <<"[READ in " << addr << "][F]" <<" insDis:"<<insDis<< s
    removeRegTainted(reg_r);
  }
}
```

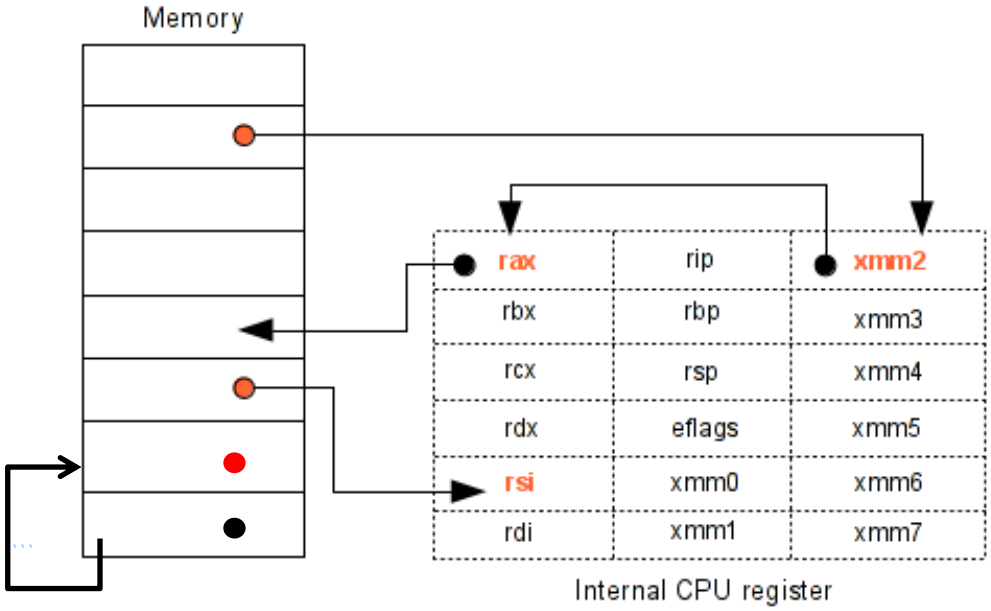
Hook Store Instruction

```
VOID WriteMem(UINT32 insAddr, std::string insDis, UINT32 OperandCount,REG reg_r, REG reg_0,UINT32 memOp, UINT32 sp)
{
  list<UINT32>::iterator i;
  UINT32 addr = memOp;
  UINT32 length=0;
  if (OperandCount != 2)
    return;
  if(!REG_valid(reg_r)){
    if(REG_valid(reg_0)){
          reg_r=reg_0;
    }else{
      /*example:mov dword ptr [eax*4+0x804a080], 0xa*/
      if(insDis.find("dword ptr",0)!=string::npos){
        length=4;
      }else if(insDis.find("word ptr",0)!=string::npos){
        length=2;
      }else{
        length=1;
      }
    }
  }
  for(i = addressTainted.begin(); i != addressTainted.end(); i++){
      if (addr == *i){
        if(insAddr<=0x8049000)
                std::cout << std::hex << insAddr<< ":\t"  <<"[WRITE in " << addr << "][F]" << " insDis:"<<insDis<< s
          //std::cout << std::hex << reg_r<< std::endl;
        if (!REG_valid(reg_r) || !checkAlreadyRegTainted(reg_r))
        {
                if(REG_is_Lower8(reg_r)||REG_is_Upper8(reg_r)){
                        length=1;
                }else if(REG_is_Half16(reg_r)){
                        length=2;
                }else if(REG_is_Half32(reg_r)){
                        length=4;
                }
          removeMemTainted(addr,length);
        }
```

Spread the taint

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.

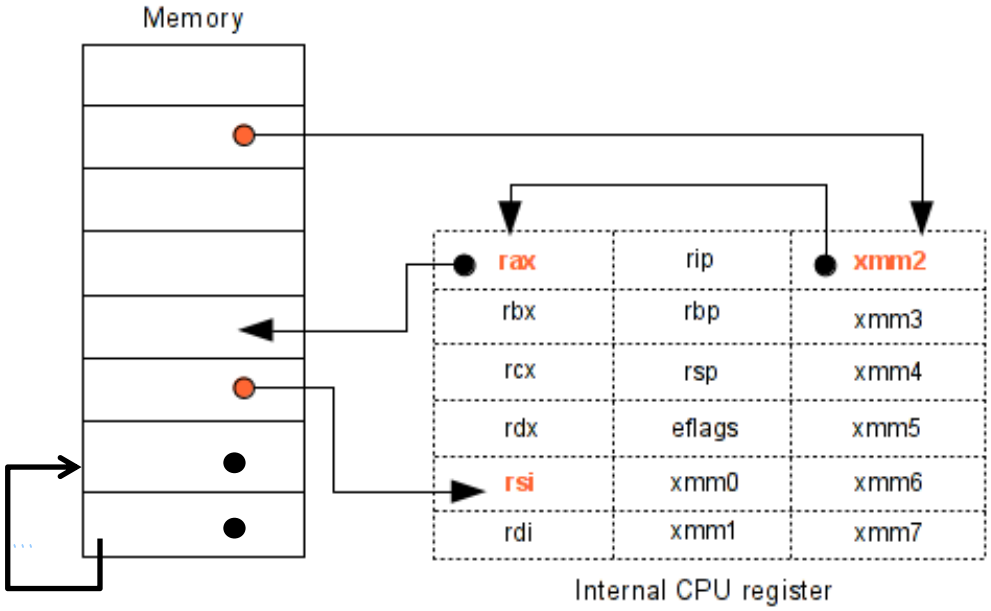
Same way, if a constant is STORED in the memory area tainted, we need to delete the taint because the user can't control this
memory location anymore.



Spread the taint

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.



Spread the taint

```
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);
}
```

Spread the taint

```
VOID spreadRegTaint(UINT32 insAddr, std::string insDis, UINT32 OperandCount,REG reg_r, REG reg_w)
{

  if (REG_valid(reg_w)){
    if (checkAlreadyRegTainted(reg_w) && (!REG_valid(reg_r) || !checkAlreadyRegTainted(reg_r))){
      if(insAddr<=0x8049000)
        std::cout <<std::hex <<insAddr<< ":\t"  << "[SPREAD][F]" << " insDis:"<<insDis<< std::endl;
      removeRegTainted(reg_w);
    }
    else if (!checkAlreadyRegTainted(reg_w) && checkAlreadyRegTainted(reg_r)){
      if(insAddr<=0x8049000)
          std::cout <<std::hex <<insAddr<< ":\t"  <<"[SPREAD][T]" <<" insDis:"<<insDis<< std::endl;
      taintReg(reg_w);
    }
  }
}
```

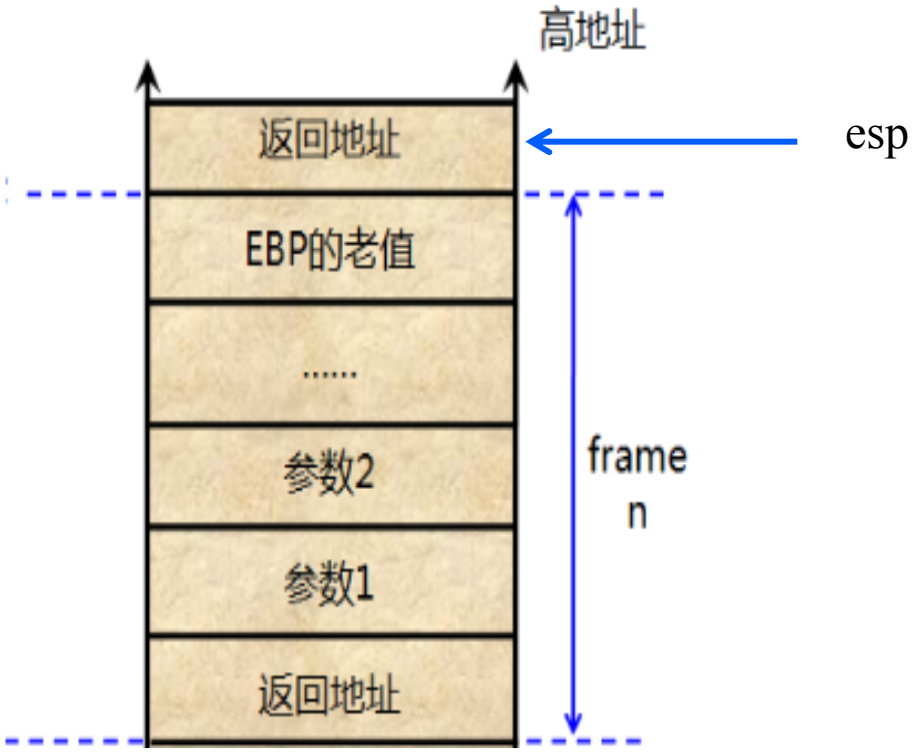
Taint analysis for security

- Detect Overflow-Return-Address

Detect overflow-Return-Value

- How to check if the return address is overflowed?
- How to get the esp value pointed to return address?

Check the return address



Before every return

pop %esp

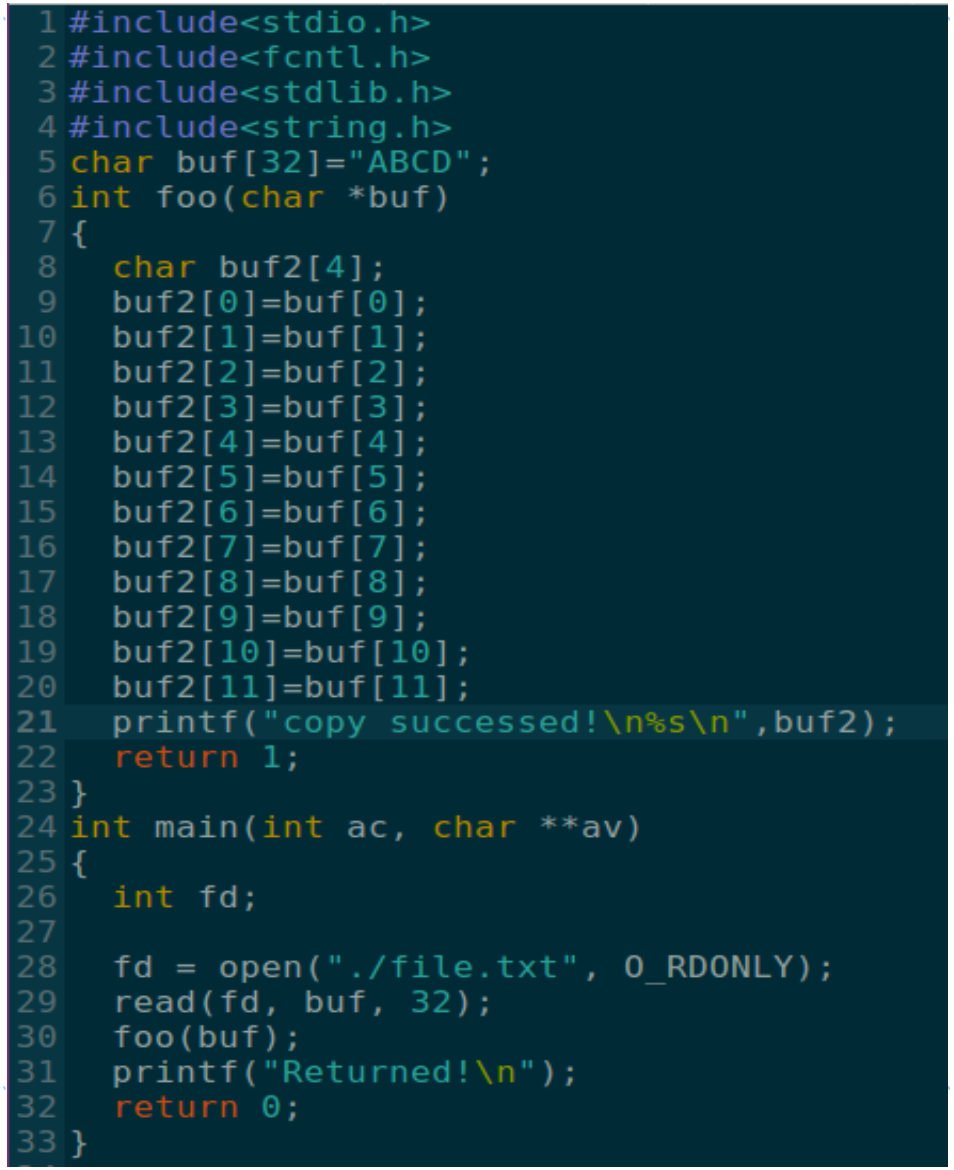
if %esp tainted

ret addr overflowed

Get esp value

instrument → get cpu context → get esp value → check the esp value

```
VOID FunctionRet(INS ins, CONTEXT *ctxt)
{
    ADDRINT value;
    UINT8 *ESP_value=(UINT8 *)&value;
    PIN_GetContextRegval(ctxt,REG_ESP,ESP_value);
    //value=value+8;
    value=value%(0x10000000);
    list<UINT64>::iterator i;
    //std::cout << std::hex <<"ESP->" << value << std::endl;
    for(i = addressTainted.begin(); i != addressTainted.end(); i++){
      if (value == *i){
        std::cout << std::hex <<"ERROR:RET Address Is Tainted!" <<" &&ESP->" << value<< std::endl;
        return;
      }
    }
}
```


#include<stdio.h>
#include<fcntl.h>
#include<stdlib.h>
#include<string.h>
char buf[32]="ABCD";
int foo(char *buf)
{
char buf2[4];
buf2[0]=buf[0];
buf2[1]=buf[1];
buf2[2]=buf[2];
buf2[3]=buf[3];
buf2[4]=buf[4];
buf2[5]=buf[5];
buf2[6]=buf[6];
buf2[7]=buf[7];
buf2[8]=buf[8];
buf2[9]=buf[9];
buf2[10]=buf[10];
buf2[11]=buf[11];
printf("copy successed!\n%s\n",buf2);
return 1;
}
int main(int ac, char **av)
{
int fd;
fd = open("./file.txt", O_RDONLY);
read(fd, buf, 32);
foo(buf);
printf("Returned!\n");
return 0;
}

Example

```
[READ in f90c970]       8048500: mov eax, dword ptr [edx+0x8]
                        eax is now freed
[READ in 804a068]       8048503: movzx eax, byte ptr [eax+0x8]
                        eax is now tainted
[WRITE in f90c964]      8048507: mov byte ptr [edx-0x4], al
                        f90c964 is now tainted
[READ in f90c970]       804850a: mov eax, dword ptr [edx+0x8]
                        eax is now freed
[READ in 804a069]       804850d: movzx eax, byte ptr [eax+0x9]
                        eax is now tainted
[WRITE in f90c965]      8048511: mov byte ptr [edx-0x3], al
                        f90c965 is now tainted
[READ in f90c970]       8048514: mov eax, dword ptr [edx+0x8]
                        eax is now freed
[READ in 804a06a]       8048517: movzx eax, byte ptr [eax+0xa]
                        eax is now tainted
[WRITE in f90c966]      804851b: mov byte ptr [edx-0x2], al
                        f90c966 is now tainted
[READ in f90c970]       804851e: mov eax, dword ptr [edx+0x8]
                        eax is now freed
[READ in 804a06b]       8048521: movzx eax, byte ptr [eax+0xb]
                        eax is now tainted
[WRITE in f90c967]      8048525: mov byte ptr [edx-0x1], al
                        f90c967 is now tainted
```

```
ERROR:RET Address Is Tainted! &&ESP->f90c964
```

Reference:

PIN introduce
[1]https://software.intel.com/sites/landingpage/pintool/docs/71313/Pin/html/index.html

PIN的API文档：
[2]https://software.intel.com/sites/landingpage/pintool/docs/71313/Pin/html/group__PIN__SYSCALL__API.html

PIN tool 下载：
[3] https ://software.intel.com/en-us/articles/pin-a-binary-instrumentation-tool-downloads

[4] FreeSentry: Protecting Against Use-After-Free Vulnerabilities Due to Dangling Pointers