TREE Taint Analysis Test Framework Design and Test Suite Development

Revision History:

Revision Number	Description	Author	Timestamp
V0.1	Initial Draft	Nathan Li	05/28/2013
V0.2	Update Some Transformation Examples	Nathan Li	06/14/2013

Abstract:

This document describes the overall idea of taint analysis, explains the main techniques used by Taint Analysis (TA) component in our Taint-enabled Reverse Engineering Environment (TREE) with illustrations of simple examples. The document first provides general background of taint analysis, then explains some design trade-offs of TREE taint analysis. It is always a difficult task to validate the results of the TA component, so the second part of the document focuses on the design of a testing framework, which can be semi-automated. The document also includes some baseline examples of a test suite. A comprehensive test suite is needed to provide high coverage of a good mix of x86 instruction set, including an extensive unit-level, highly crafted "transformation-function" style programs and a small number representative, real programs.

Contents

1 Overview of Taint Analysis

Simply speaking, taint analysis is to track information flow inside a program, during a program execution. The information to track can be anything, from function or system call parameter to a register, or a memory location. Among them, input is one of the most interesting information. So we will use input (taint) in our example in this document.

1.1 Main steps of Taint Analysis at Binary Level

Taint analysis can work at source code, interpreter or binary level; but binary level taint analysis tool fits security applications particularly well since all programs run eventually in the form of machine code and binary executable includes code not included in source code.

There are basically three steps involved in a complete taint analysis cycle: taint marking, taint tracking and taint checking. The three steps are shown in Figure 1:

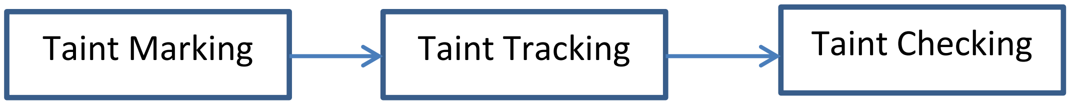


Figure 1 Steps of Taint Analysis

2 Taint Analysis in TREE

Taint analysis can be done either online or offline. Online taint analysis marks initial taint (*taint source*), tracks taint and checks taint at specific point (*taint sink*) all at the same time the target program executes; offline taint analysis works on an *execution trace* that captures the program states from the previous execution of the target program. TREE uses offline taint analysis because it fits better into the overall TREE system design, Figure 2.

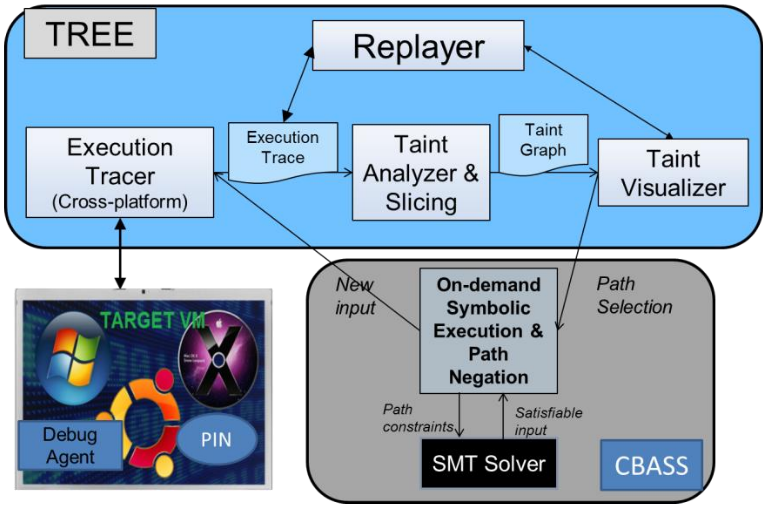


Figure 2 TREE Architecture

2.1 Trace Generation

TREE can produce execution trace through either our PIN Execution Tracer plug-in (exetrace.dll) or our automated IDA Tracer. PIN execution tracer can produce smaller encoded binary trace and faster – a good option for user who is familiar with Dynamic Instrumentation Framework (DIB) and PIN framework. IDA Trace is directly available as an IDA plug-in that can produce text-format trace, which user can inspect manually; and IDA Tracer comes with an intuitive GUI to guide user through trace generation. TREE also provides a PinAgent program to bridge TREE GUI to our PIN Execution Tracer so user doesn't have to deal with tedious details to get PIN execution trace.

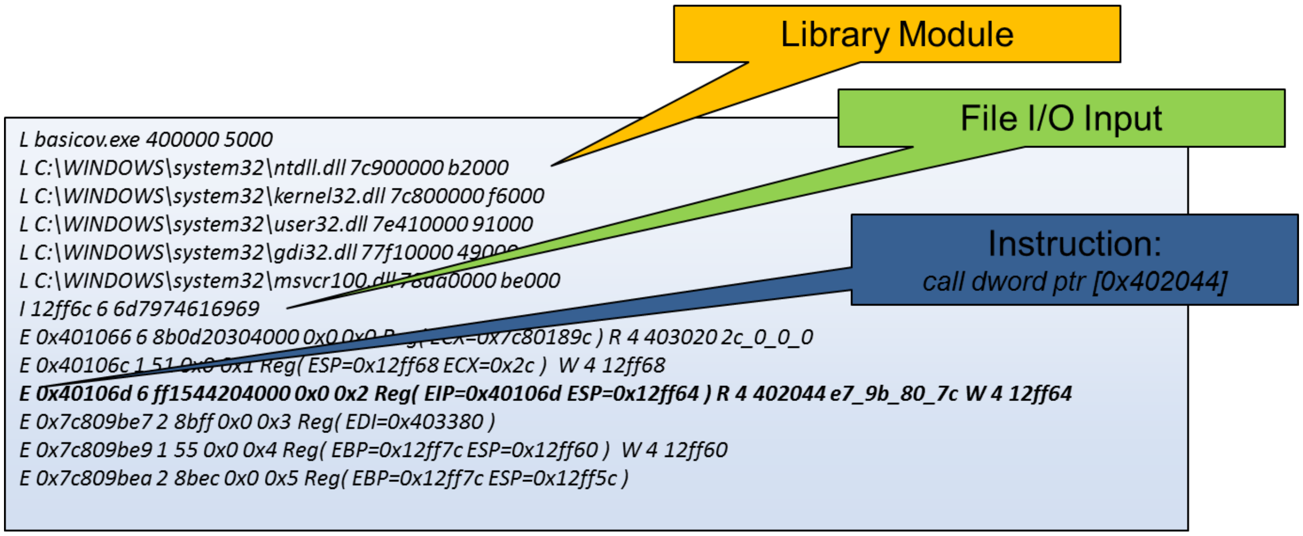
2.1.1 Trace generation from PIN plug-in

TREE Pin Tracer generates compressed binary trace through Pin Plugin.

2.1.2 Trace generation from IDA debugging Tracer

TREE Debug Tracer uses GUI to guide user to generate text format trace.

2.2 Trace Parsing and Initial Taint Source Marking



```
L basicov.exe 400000 5000
L C:\WINDOWS\system32\ntdll.dll 7c900000 b2000
L C:\WINDOWS\system32\kernel32.dll 7c800000 f6000
L C:\WINDOWS\system32\user32.dll 7e410000 91000
L C:\WINDOWS\system32\gdi32.dll 77f10000 49000
L C:\WINDOWS\system32\msvcr100.dll 78aa0000 be000
I 12ff6c 6 6d7974616969
E 0x401066 6 8b0d20304000 0x0 0x0 Reg( ECX=0x7c80189c ) R 4 403020 2c_0_0_0
E 0x40106c 1 51 0x0 0x1 Reg( ESP=0x12ff68 ECX=0x2c )  W 4 12ff68
E 0x40106d 6 ff1544204000 0x0 0x2 Reg( EIP=0x40106d ESP=0x12ff64 ) R 4 402044 e7_9b_80_7c W 4 12ff64
E 0x7c809be7 2 8bff 0x0 0x3 Reg( EDI=0x403380 )
E 0x7c809be9 1 55 0x0 0x4 Reg( EBP=0x12ff7c ESP=0x12ff60 )  W 4 12ff60
E 0x7c809bea 2 8bec 0x0 0x5 Reg( EBP=0x12ff7c ESP=0x12ff5c )
```

TREE Trace captures most of the program states:

- Captures a snapshot of the program state at the beginning
- Tracks all instruction level state delta(not the whole state)
- Tracks only relevant read/write memory access (address and value)
- Tracks only relevant register changes and values
- Trace generation is fully automated, no user involvement needed
- PIN Trace records Input/Output buffers for System Call

Initial taint source can be marked through GUI; otherwise user input recorded in the trace is used as initial taint.

2.3 Taint Tracking

TREE tracks taint through a combination of static taint template/category and instruction-specific propagation.

The static taint template and category is generated using the x86Decoder (based on XED) we developed.

2.3.1 Static x86 instruction taint template

TREE provides static x86 instruction taint template through x86 decoder, by modifying Xed [2] examples.

All Intel Architecture instructions are encoded using subsets of the general machine instruction format Instructions consist of optional instruction prefixes (in any order), primary opcode bytes (up to three bytes), an addressing-form specifier (if required) consisting of the ModR/M byte and sometimes the SIB (Scale-Index-Base) byte, a displacement (if required), and an immediate data field (if required).

Inputs to the x86 decoder:

-- Instruction length

-- Instruction encoded bytes

Output:

-- A structure to describe the taint relation between instruction source, destination operands, including explicit and implicit operands. The structure also contains each operand's type (immediate, register or memory), width in bits, read/write attributes.

An example:

Inst_category=4, Disassembly: sub $0xc, %esp

src_operand_num=2:

width=32, rw=1, type=2, ea_string=ESP

width=8, rw=2, type=1, ea_string=c

dest_operand_num=2:

width=32, rw=1, type=2, ea_string=ESP

width=32, rw=3, type=2, ea_string=EFLAGS[of sf zf af pf cf]

Another example:

Inst_category=4, Disassembly: addl -0xc(%ebp), %eax

src_operand_num=2:

width=32, rw=1, type=2, ea_string=EAX

width=32, rw=2, type=3, ea_string=SEG=SS:BASE=EBP:DISP=-12

dest_operand_num=2:

width=32, rw=1, type=2, ea_string=EAX

width=32, rw=3, type=2, ea_string=EFLAGS[of sf zf af pf cf]

2.3.2 X86 Instruction Categorization and Taint by Category

We roughly classify x86 instructions into 40 categories; of the 40 categories, we focus on the following most commonly used categories and some typical example instructions.

- DATAXFER:mov, movzxw,movb,
- BINARY: cmp, sub, add, inc, dec,
- STRINGOP: rep movsdl, rep stosdl,
- COND_BR: jnz, jl, jz
- LOGICAL: xor, test, or, and, not,
- SHIFT: shr, shl,
- FLAGOP: cld, std,
- PUSH: push
- POP: pop
- RET: ret
- CALL: call
- MISC: leavel, lea,

2.3.3 Taint Precision:

TREE tracks taint at byte level.

Precision may be traded off when complexity or data volume is too high.

- 1 To 1:
    - Examples:
        - mov %eax,%esi, <u>esi <- eax,</u>
        - push esi, mem[esp] <- esi
        - pop ebp ebp <-mem[esp]
- M to N:
    - Examples:
        - add %ecx, %esi: esi < esi, ecx
        - *addl -0xc(%ebp), %eax:*
        - *eax<- eax, mem[ebp-0xc]*
        - *EFLAGS[of sf zf af pf cf] <- eax, mem[ebp-0xc]*

2.3.4 Taint Correctness: Over-Taint and Under-Taint

- **Over-taint: Something got tainted when it should not be:**
    - xor eax, eax: eax <-0 regardless if eax was tainted or not
    - and 0x0, eax : eax <-0 regardless if eax was tainted or not

  Detaint when necessary

- **Under-taint: Something is not tainted when it should be**
    - if (x != 0) y = 1 else y = 0; y is dependent on x, but y doesn't directly affected by x

Under-taint is usually because of control-flow dependency, which we can ignore for now

2.4 Taint Checking and Security Analysis

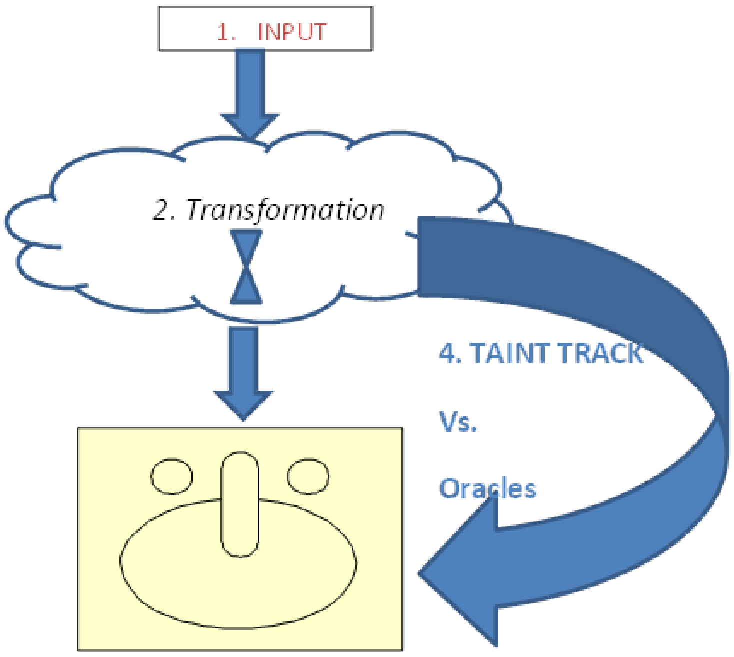
What to check for taint checking is highly dependent on specific tasks, vulnerability, exploitation or malware behavior. For our taint analyzer testing purpose, we use a basic overflow scenario that some input bytes overwrite the EIP target; we provide different transformation functions to transform input, and validate if TREE taint analyzer can attribute to the right part of the input.

3 TREE Taint Analysis Testing Framework

3.1 Overview of the Testing Framework:

The framework, shown in Figure 1, has four main steps:

1. Use Standard ReadFile to read input and mark them as initial taints
2. Develop Transformation Functions to transform these inputs
3. Develop Oracles for the Transformation Functions developed in Step 2
4. Compare TA analysis results with Oracles to see if they match; if not, why?
5. If there's a bug in Taint Track, fix it and go back to step 1.



3.2 X86 Instruction Coverage and the Priority List

- DATAXFER:mov, movzxw,movb,
- BINARY: cmp, sub, add, inc, dec,
- STRINGOP: rep movsdl, rep stosdl,
- COND_BR: jnz, jl, jz
- LOGICAL: xor, test, or, and, not,
- SHIFT: shr, shl,
- FLAGOP: cld, std,
- PUSH: push
- POP: pop
- RET: ret
- CALL: call
- MISC: leavel, lea,

3.3 A working template: BasicOV

```
int main(int argc, char* argv[])
{
	char sBigBuf[16]={0};

	hFile = CreateFile("mytaint.txt",      // Open One.txt
                              GENERIC_READ,           // Open for reading
                              0,                      // Do not share
                              NULL,                   // No security
                              OPEN_EXISTING,          // Existing file only
                              FILE_ATTRIBUTE_NORMAL,  // Normal file
                              NULL);                  // No template file

	if (hFile == INVALID_HANDLE_VALUE)
	{
```

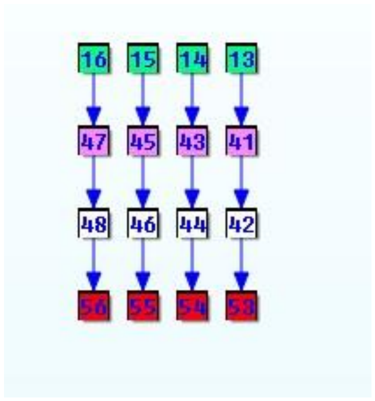
```
                return 1;
        }

        DWORD dwBytesRead;
        ReadFile(hFile, sBigBuf, 16, &dwBytesRead, NULL);

        Transformation Function is EMPTY Here

        StackOVflow(sBigBuf,dwBytesRead);// Effect

        return 0;
}
void StackOVflow(char *sBig,int num)
{
        char sBuf[8]= {0};

        for(int i=0;i<num;i++)
        {
                sBuf[i] = sBig[i];
        }
        return;
}
```

Input 1-8 bytes goes to sBuf, 9-12 bytes to overwrite EBP, 13-16 bytes to overwrite EIP

Taint Track Result: Taint Graph

```
[53]reg_eip_0_40032[0xd5:40032]<-retl  {D}42
[42]mem_0x3afc8c[0xa7:40032]<-movb  %dl, -0x8(%ebp,%ecx,1){D}41
[41]reg_edx_0_40032[0xa6:40032][0xac:40032]<-movb  (%eax), %dl{D}13
[13]mem_0x3afca8[-0x1:-1]
[54]reg_eip_1_40032[0xd5:40032]<-retl  {D}44
[44]mem_0x3afc8d[0xb3:40032]<-movb  %dl, -0x8(%ebp,%ecx,1){D}43
[43]reg_edx_0_40032[0xb2:40032][0xb8:40032]<-movb  (%eax), %dl{D}14
[14]mem_0x3afca9[-0x1:-1]
[55]reg_eip_2_40032[0xd5:40032]<-retl  {D}46
[46]mem_0x3afc8e[0xbf:40032]<-movb  %dl, -0x8(%ebp,%ecx,1){D}45
[45]reg_edx_0_40032[0xbe:40032][0xc4:40032]<-movb  (%eax), %dl{D}15
[15]mem_0x3afcaa[-0x1:-1]
[56]reg_eip_3_40032[0xd5:40032]<-retl  {D}48
[48]mem_0x3afc8f[0xcb:40032]<-movb  %dl, -0x8(%ebp,%ecx,1){D}47
[47]reg_edx_0_40032[0xca:40032][0xd0:40032]<-movb  (%eax), %dl{D}16
[16]mem_0x3afcab[-0x1:-1]
```

Graphically, the taint relation can be represented in a graph, like the following:



3.4 Transformation function and the Oracle:

3.4.1 Transformation suggestions:

- Oracle: derived from your design or debugging
- Compile transformation function in different compiler settings/optimizations
- Use different compilers, like gcc, and different optimization settings to compile

3.4.2 A couple of example:

BasicOV_Memset program:

```
void StackOVflow(char *sBig,int num)
{
       char sBuf[8]= {0};
       strcpy(sBuf, sBig);
       memset(sBuf, 0, num-1);
//ONLY the last byte of input should taint one byte
of EIP; the other bytes are zerod out.
       return;
}
```

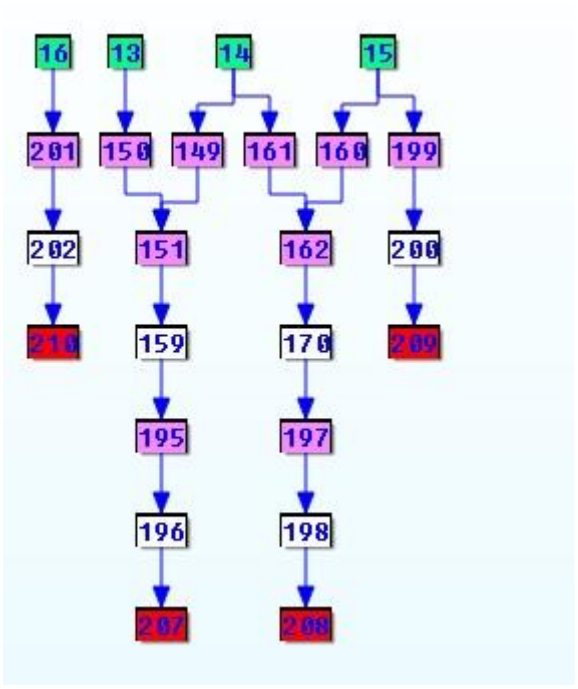
The taint result:

```
[178]reg_eip_3_0[0xd0:0x0]<-retl  {D}161

[161]mem_0x12ff5f[0x7e:0x0]<-movl  %edx, (%edi){D}157

[157]reg_edx_3_0[0x7a:0x0]<-movl  (%ecx), %edx{D}16

[16]in_0x12ff7b[0x0:0x0]<-0xffff:ReadFile
```

BasicOV_Plus program:

```
ReadFile(hFile, sBigBuf, 16, &dwBytesRead, NULL);

CloseHandle(hFile);

for(int i=0; i< (dwBytesRead-2); i++)
        sBigBuf[i] +=sBigBuf[i+1];

StackOVflow(sBigBuf,dwBytesRead);
```

The taint result:



* More examples are available under TA_TestSuite folder:

4 Test Suite:

4.1 Plan to expand (transformation-function) unit level test suites for the popular instruction categories:

- 10 transformation functions for each category
- 10 transformation functions cover the variants of the instructions
- Transformation functions to be compiled with different compilers(VC, gcc, llvm) and on Windows/Linux platforms

4.2 Real world programs

- Benign program or malware

5 References:

[1] *Intel Architecture Software Developer's Manual, particularly Volume 2" Instruction Set Reference*

[2] *XED. Available online as of 06/14/2013. http://www.pintool.org/downloads.html*