Taint-enabled Reverse Engineering Environment (TREE) Design Document

Revision History:

Revision Number	Description	Authors	Timestamp
V0.1	Initial Draft	Nathan Li, Xing Li, Loc Nguyen	06/27/2013
V0.2	Update	Xing Li/Nathan Li	07/09/2013

Abstract: This document describes the overall idea and architecture of Taint-enabled Reverse Engineering Environment (TREE) and detailed designs of its main components: TREE Tracer, TREE Analyzer, and TREE Visualizer. The development of TREE is changing rapidly, and this document will update frequently. Check often for latest update at http://code.google.com/p/tree-cbass/.

This document is for users who intend to learn the designs and internals behind the tool. For installation, features and usages of TREE, please check "TREE User Manual".

1 Table of Contents

2 TREE System Overview

TREE (Tainted-enabled Reverse Engineering Environment), is the front-end of our cross-platform interactive analysis framework, which integrates state-of-the-art dynamic analysis techniques with a mainstream reverse engineering tool(IDA Pro) to meet the demand in security practice. Our framework, also comprises the back-end CBASS (Cross-platform Binary Automated Symbolic execution System) to support interactive analysis through on-demand symbolic execution. We will present CBASS in separate document and focus this document on TREE,

Figure 1 shows the architecture of our interactive analysis system and the main components.

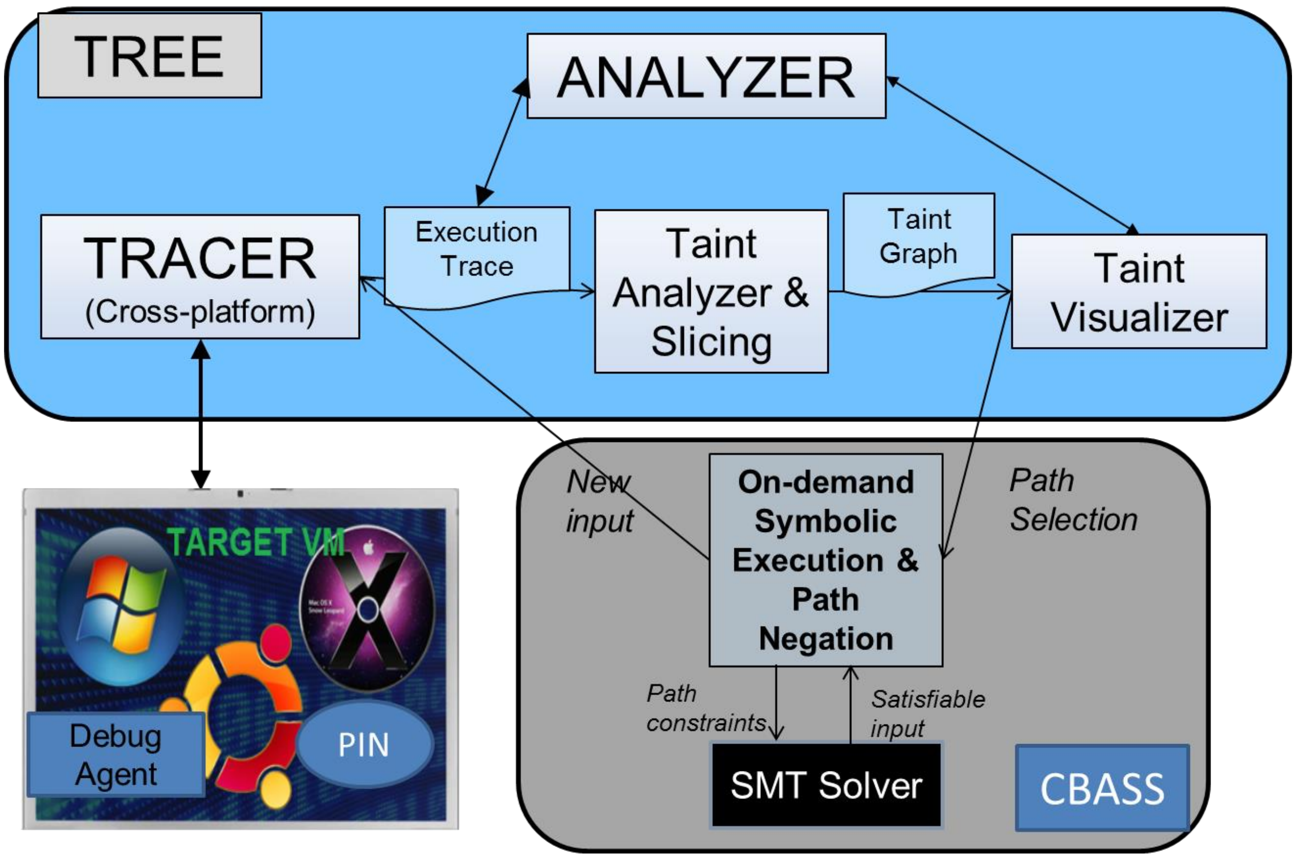


Figure 1 - TREE System Architecture

3 TREE Tracer

3.1 Overview of Tracing

Program tracing is the first step and serves as the basis of further analysis. Detail instruction tracing enables sophisticated binary analysis such as taint analysis and symbolic execution. Besides bug hunting, tracing can be useful to understand a program's control and data flow. There are many different ways to generate execution traces.

3.2 Tracing with debuggers

Many debuggers offer the ability to trace a program's execution. Debuggers such as GDB, OllyDbg, and IDA Pro use breakpoint-based mechanism, which is supported by almost all processors and operating systems. Tracing with debuggers is relatively slower, but it provides a unified approach for collecting execution traces from different platfroms.

3.3 Tracing with system tracers

Strace, Dtrace, LTTng, and SystemTap are examples of system tracers. These system tracers are embedded inside the operating system. They offer good ways to monitor system calls and signals by the program. The downside of using system tracers is that they do not offer much flexible in tracing. The tracing is restricted to system calls and signals. A user cannot define arbitrary points in a program to trace.

3.4 Tracing with instrumentation tools

Pintool, DynamoRio, and Valgrind are binary instrumentation frameworks that can also be used to generate execution traces. These tools can be used perform program analysis in runtime or generate traces for offline analysis. Custom plugins can be leveraged to monitor and generate program execution. Depending on the configuration of these tools, the performance of these tracers are often much better than using debuggers for tracing, however, DBI and their plugins are often platform (OS and instruction set) specific.

3.5 Tracing with TREE

The TREE Tracer plug-in for IDA Pro is built around the IDA Pro debugging framework. Although the performance of our tracer is slower than tracing with instrumentation tools, because of the debugging framework, we believe the flexibility, portability, and power of IDA Pro outweighs its performance disadvantage. We also offer Pintool integration with our TREE Tracer plug-in to take advantage of both speed and power of IDA and Pintool.

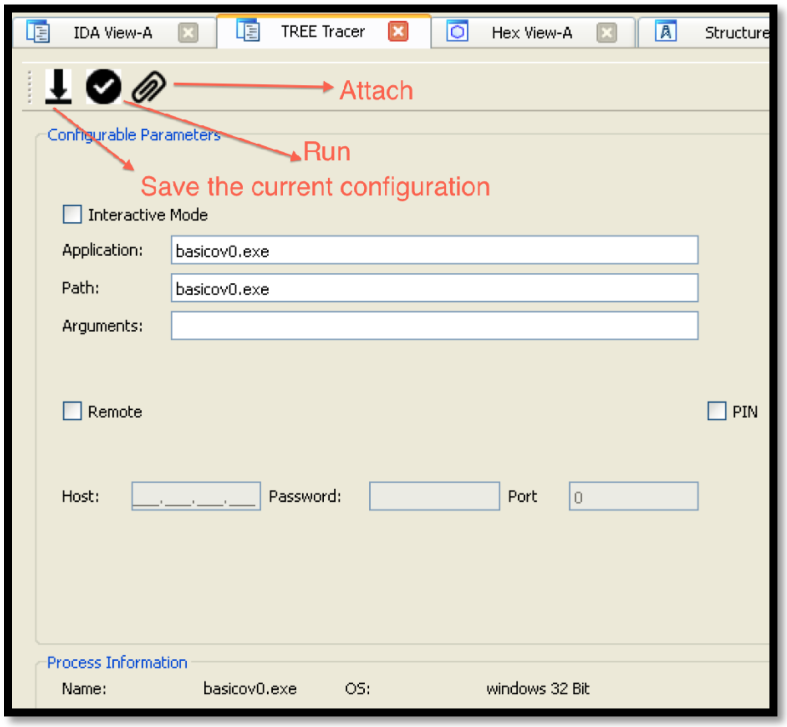
The main advantage of using this architecture, besides running within the most popular dissembler, is the ability to leverage IDA Pro's vast debugging features. With IDA Pro, the user can debug local or remote programs executing in kernel mode, user mode, on Android, Windows, Linux, and 32 bit or 64 bit mode. IDA Pro runs on Windows, Linux, and MacOSX.

For a complete list of IDA Pro debuggers please visit the Hex-Rays website.

https://www.hex-rays.com/products/ida/debugger/

3.5.1 TREE Tracer Plugin

The TREE Tracer plugin was developed using IDAPython and PySide. One of the main design goals of the TREE Tracer GUI was for IDA Pro users to feel comfortable using TREE. The GUI allows the user to interface either with the IDA Debugger or Pintool. Figure 2 shows the TREE Tracer GUI. The TREE Tracer GUI is very similar to the IDA Debugger process options Figure 3, so if the user is familiar with IDA, he/she will feel comfortable with the TREE Tracer GUI. The user can change and save the current configuration. Tracing can be done either by clicking the run button to start the program from the TREE Tracer or by attaching to an existing program by clicking the attach button.



Interactive Mode [ ]

Application: basicov0.exe

Path: basicov0.exe

Arguments:

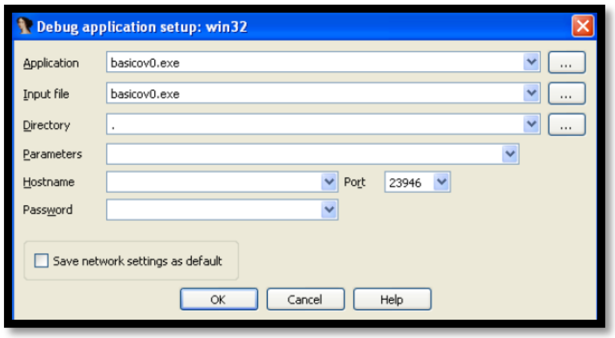
Remote [ ]

PIN [ ]

Host: ___.___.___.___ Password: Port 0

Name: basicov0.exe OS: windows 32 Bit

Figure 2 - TREE Tracer GUI



Application: basicov0.exe

Input file: basicov0.exe

Directory: .

Parameters:

Hostname: Port 23946

Password:

Save network settings as default [ ]

Figure 3 - IDA Pro Debugger Process Options

3.5.2 Saving configuration settings

With the TREE Tracer GUI, the user can change and save the configuration settings. The configuration settings are stored in “C:\Documents and Settings\[current user]\My Documents\TREE\config.xml” for Windows XP and “C:\Users\[current user]\Documents\TREE\config.xml” for Windows 7.

3.5.3 Run

With this feature, the TREE Tracer will start the tracing process from creating a new process.

3.5.4 Attach

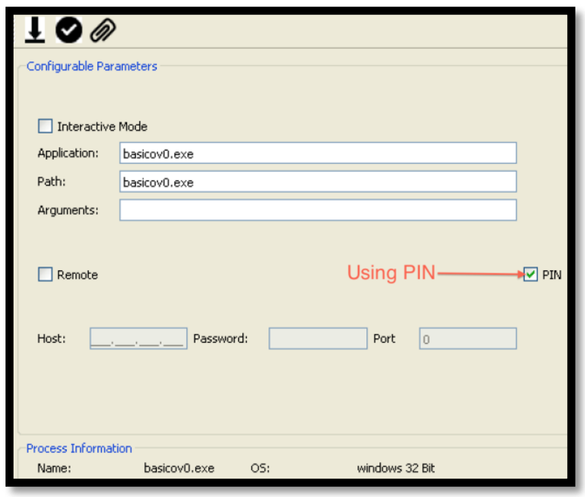
With this feature, the TREE Tracer will attach to a running process. The process attach feature only works in interactive mode. Please see Interactive Mode

3.6 Trace Generation with the TREE Tracer

The IDA Tracer comes with an intuitive GUI to guide the user through trace generation. The user has a choice to use either integrated PinAgent for PIN-based trace or the IDA Debugger to generate a trace.

3.6.1 Using the PinAgent

The TREE Tracer GUI communicates with an active PinAgent to generate the trace.



Configurable Parameters

☐ Interactive Mode

Application:	basicov0.exe
Path:	basicov0.exe
Arguments:	

☐ Remote ☑ PIN

Host: ___.___.___.___ Password: Port 0

Process Information

Name: basicov0.exe OS: windows 32 Bit

Figure 4 - TREE Tracer with PinAgent

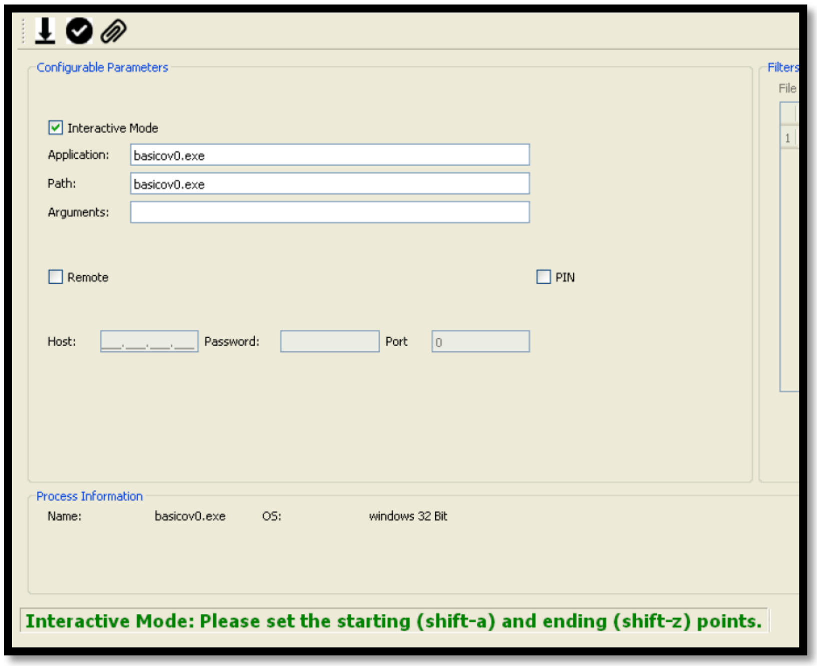
- *This feature is still being developed. We hope to have it ready for our next release.*

3.6.2 Using the IDA Debugger

The TREE Tracer can run in either interactive or non-interactive mode. Interactive mode allows the user freedom to trace any portion of the disassembly. Non-interactive mode uses filters and API monitoring to generate traces.

3.6.3 Interactive Mode

In this mode, the user clicks on the “Interactive Mode” check box.


Configurable Parameters

☑ Interactive Mode

Application: basicov0.exe

Path: basicov0.exe

Arguments:

☐ Remote ☐ PIN

Host: ___.___.___.___ Password: Port 0

Process Information

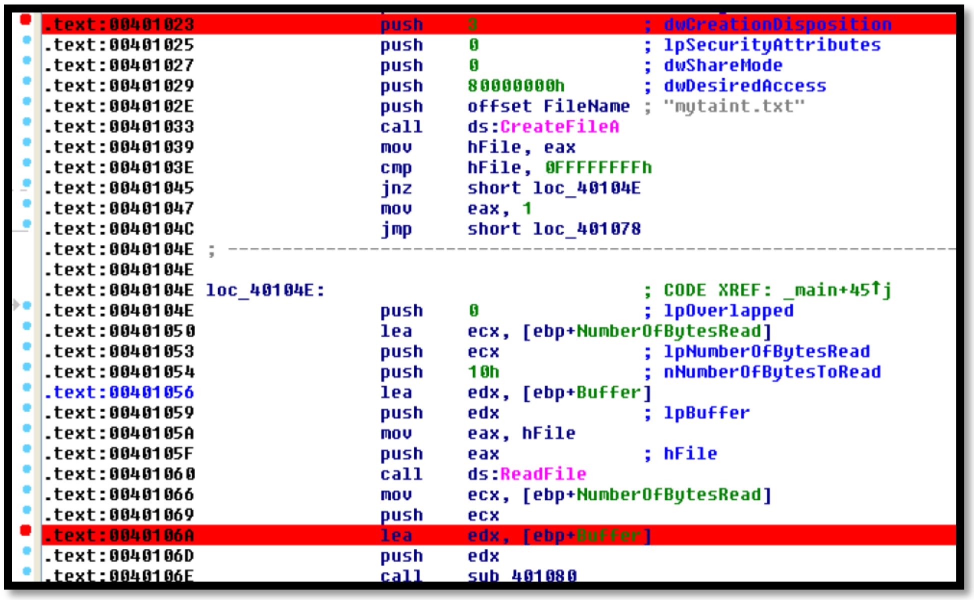
Name: basicov0.exe OS: windows 32 Bit

Interactive Mode: Please set the starting (shift-a) and ending (shift-z) points.

Figure 5 - Interactive Mode

Next, the user selects the starting point to trace by placing the cursor at an address in the IDA View disassembly Window and holding down both the shift and a key. This is where the TREE Trace will start tracing. In this example, it is the address at ".text:00401023 push 3".

Next, the user selects the ending point to the trace by placing the cursor at an address in the IDA View disassembly Window and holding down both the shift and z key. This is where the TREE Trace will stop tracing. In this example, it is the address at ".text:0040106A lea edx, [ebp+Buffer]"



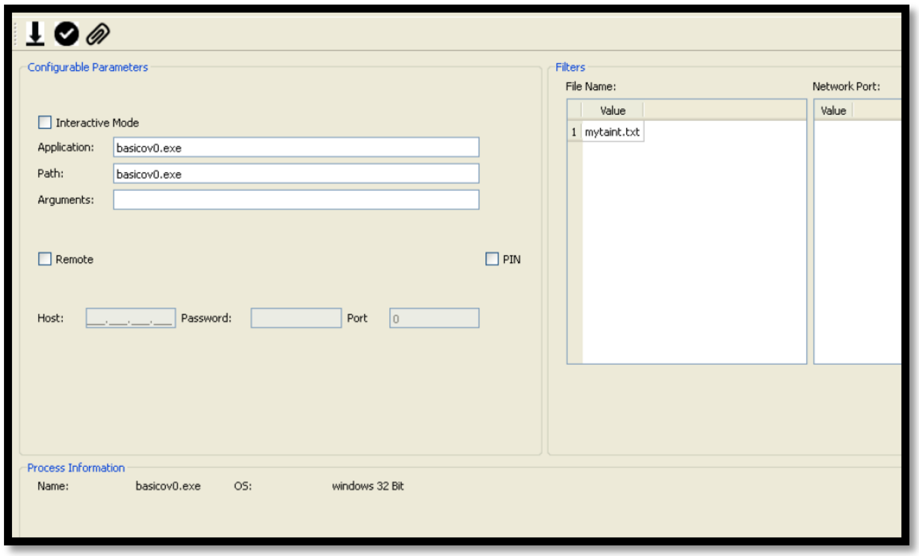
```
.text:00401023                 push    3               ; dwCreationDisposition
.text:00401025                 push    0               ; lpSecurityAttributes
.text:00401027                 push    0               ; dwShareMode
.text:00401029                 push    80000000h       ; dwDesiredAccess
.text:0040102E                 push    offset FileName ; "mytaint.txt"
.text:00401033                 call    ds:CreateFileA
.text:00401039                 mov     hFile, eax
.text:0040103E                 cmp     hFile, 0FFFFFFFFh
.text:00401045                 jnz     short loc_40104E
.text:00401047                 mov     eax, 1
.text:0040104C                 jmp     short loc_401078
.text:0040104E ; ---------------------------------------------------------------------------
.text:0040104E
.text:0040104E loc_40104E:                             ; CODE XREF: _main+45↑j
.text:0040104E                 push    0               ; lpOverlapped
.text:00401050                 lea     ecx, [ebp+NumberOfBytesRead]
.text:00401053                 push    ecx             ; lpNumberOfBytesRead
.text:00401054                 push    10h             ; nNumberOfBytesToRead
.text:00401056                 lea     edx, [ebp+Buffer]
.text:00401059                 push    edx             ; lpBuffer
.text:0040105A                 mov     eax, hFile
.text:0040105F                 push    eax             ; hFile
.text:00401060                 call    ds:ReadFile
.text:00401066                 mov     ecx, [ebp+NumberOfBytesRead]
.text:00401069                 push    ecx
.text:0040106A                 lea     edx, [ebp+Buffer]
.text:0040106D                 push    edx
.text:0040106E                 call    sub_401080
```

Figure 6 - IDA Pro's Disassembly View

3.6.3.1 *Non-Interactive Mode*

In this mode, the TREE Tracer handles when to start and stop tracing. The TREE Tracer monitors when a file or network port is open and read from. The user can set up file and/or network filters to control which file or network port to trace.

In the example below, the TREE Tracer will start the program called "basicov.exe". The trace will start only if basicov.exe opens a file called "mytaint.txt". The actual tracing will start when basicov.exe reads from mytaint.txt and stop when basicov.exe exits or crashes.



Configurable Parameters

☐ Interactive Mode

Application:	basicov0.exe
Path:	basicov0.exe
Arguments:	

☐ Remote ☐ PIN

Host: ___.___.___.___ Password: Port 0

Filters

File Name:

	Value
1	mytaint.txt

Network Port:

Value

Process Information

Name: basicov0.exe OS: windows 32 Bit

Figure 7 Non-Interactive Mode

3.7 IDA Pro's debugging framework

IDA Pro provides the DBG_Hook API to communicate with the IDA Pro Debugger. The IDA Pro debugger will relay all of its debugging information back to TREE Tracer callback functions.

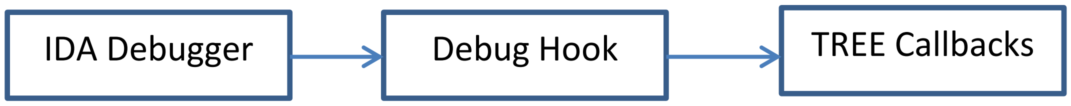
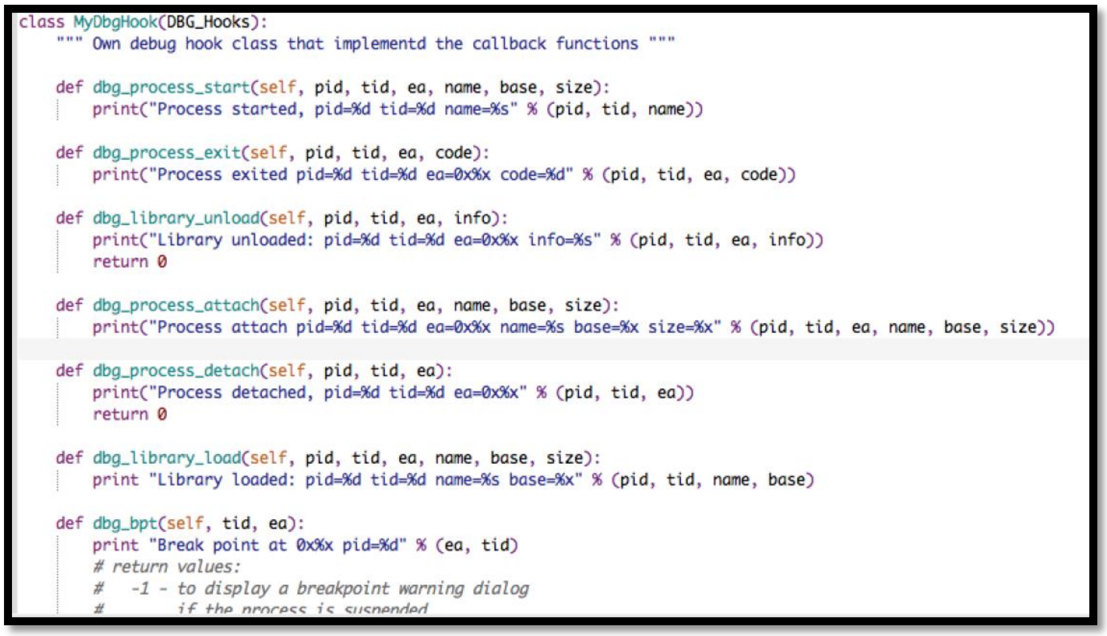


Figure 8 - IDA Pro's Debugging Framework



```
class MyDbgHook(DBG_Hooks):
    """ Own debug hook class that implementd the callback functions """

    def dbg_process_start(self, pid, tid, ea, name, base, size):
        print("Process started, pid=%d tid=%d name=%s" % (pid, tid, name))

    def dbg_process_exit(self, pid, tid, ea, code):
        print("Process exited pid=%d tid=%d ea=0x%x code=%d" % (pid, tid, ea, code))

    def dbg_library_unload(self, pid, tid, ea, info):
        print("Library unloaded: pid=%d tid=%d ea=0x%x info=%s" % (pid, tid, ea, info))
        return 0

    def dbg_process_attach(self, pid, tid, ea, name, base, size):
        print("Process attach pid=%d tid=%d ea=0x%x name=%s base=%x size=%x" % (pid, tid, ea, name, base, size))

    def dbg_process_detach(self, pid, tid, ea):
        print("Process detached, pid=%d tid=%d ea=0x%x" % (pid, tid, ea))
        return 0

    def dbg_library_load(self, pid, tid, ea, name, base, size):
        print "Library loaded: pid=%d tid=%d name=%s base=%x" % (pid, tid, name, base)

    def dbg_bpt(self, tid, ea):
        print "Break point at 0x%x pid=%d" % (ea, tid)
        # return values:
        #   -1 - to display a breakpoint warning dialog
        #        if the process is suspended
```

Figure 9 - Debug Hook, IDAPython debug hook example

4 TREE Analyzer

4.1 Overview of Taint Analysis

Simply speaking, taint analysis is to track information flow inside a program, during a program execution. Information 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.

4.2 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 in machine code eventually 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:

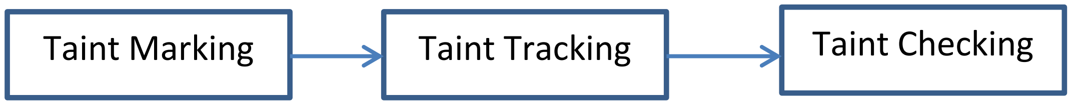
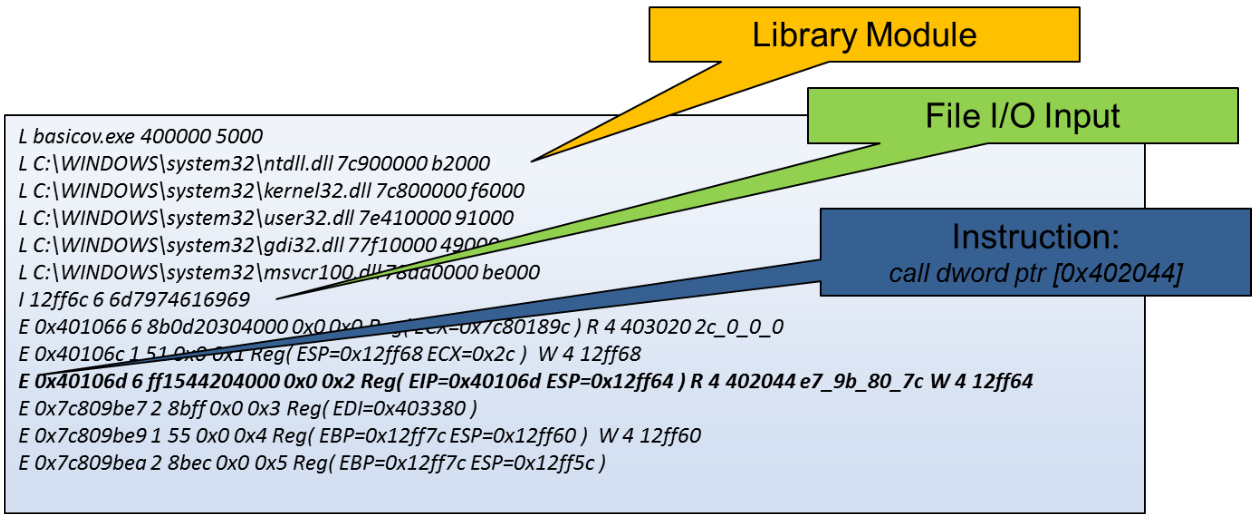


Figure 10 – Complete Taint analysis cycle

4.3 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 1.

4.4 Trace Parsing and Initial Taint Source Marking


Library Module
File I/O Input
Instruction:
call dword ptr [0x402044]

```
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 )
```

Figure 11 – Sample execution trace output

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 Replayer GUI; otherwise user input recorded in the trace is used as initial taint.

4.5 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.

4.5.1 X86 instruction static taint template

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]

- *eax<- eax, mem[ebp-0xc]*

*EFLAGS[of sf zf af pf cf] <- eax, mem[*X86 Instruction Categorization and Taint by Category

Of all the x86 instructions, we can roughly classify them into 40 categories; of the 40 categories, we focus on the following most commonly used categories:

- 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,

4.5.2 **Taint Precision:** Taint by byte

Taint Relations:

1 To 1:

- mov %eax,%esi, esi <- eax,
- push esi, mem[esp] <- esi

- pop ebp ebp <-mem[esp]

M to N:

- add %ecx, %esi: esi < esi, ecx
- *addl -0xc(%ebp), %eax:*
- *ebp-0xc]*

4.5.3 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 not directly through data movement.

4.6 Taint Checking and Security Analysis

Depending on security applications, user may choose different taint sinks. For exploits, taint sinks may registers like EIP, EBP or memory locations. To detect vulnerabilities, taint sinks are usually the unexplored branches. When a branch instruction is encountered in the trace and the branch predicate is tainted, it means the user may craft the input to steer the program run a different path, the exact value of the input bytes, however, need a constraints generator and constraints solver(like SMT Solver)to determine. We will release a version that integrates with CBASS, our symbolic execution engine, in the future.

5 TREE Taint Analysis Testing Framework

At the unit testing level, we have used a number of binary programs(eacharound100LOC)to check if the core analysis algorithms in TREE are implemented correctly. We have designed various transformation functions to process the input(taint source) and created the corresponding test oracles to ensure that TREE produces correct results. The test programs are compiled on different platforms (Windows, Linux, and Android) using different compilers(VC,GCC) with various optimization settings.

5.1 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 is a bug in Taint Track, fix it and go back to step 1.

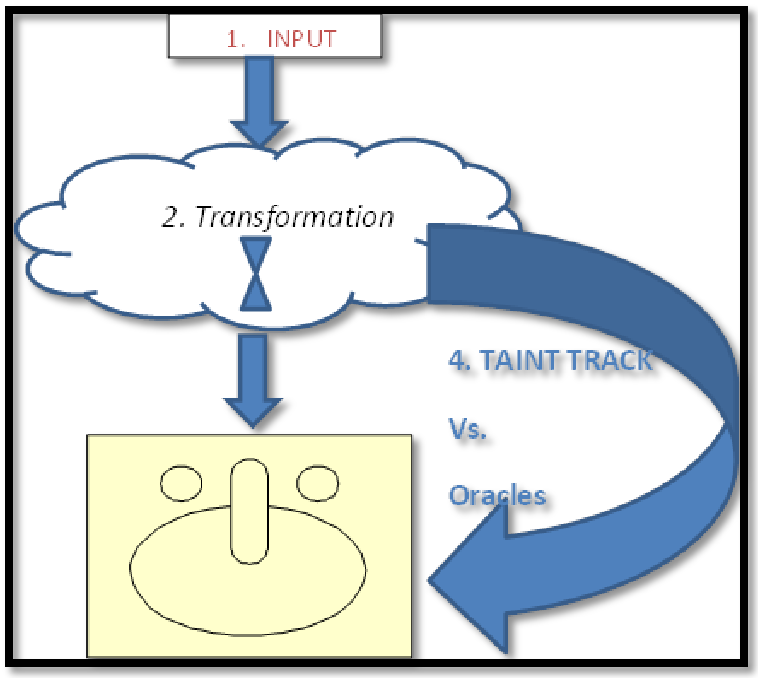


Figure 12– Transformation diagram

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

        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;
}
```

Oracle: derived from your design or debugging

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]
```

5.2 Transformation function and the Oracle:

Some suggestions for test case expansion:

a. Compile transformation function in different compiler settings/optimizations
b. Use different compilers, like gcc, and different optimization settings to compile

A variant of the *basicOV* program(*basicov_plus.exe*):

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

CloseHandle(hFile);

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

StackOVflow(sBigBuf,dwBytesRead);
```

5.3 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,

6 Test Suite:

6.1 Code snippet (transformation-function like) unit level test suites for above 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

6.2 Real world **programs**

- Evaluations are underway. We will update this section after the first round of evaluations.

7 TREE Visualizer

7.1 Overview of TREE Visualizer

The visualizer component of TREE is the interface to allow for representation of the trace and taint information in graph format. Through the graph format, the taint relationships will be depicted through edge relationships with each graph node representing a single taint.

7.2 Graph Drawing

There are many options in representing the flow of tainted data - the TREE visualizer specifically utilizes node-link diagrams. The vertices or nodes are represented as boxes and the edges are designated as a directional line indicating the transformation flow from one node state to the next.

For more information on graph drawing theory, reference the graphviz theory page.

http://graphviz.org/Theory.php

7.3 Data Design

The taint data is stored within a taint object data structure within memory that couples the taint attributes and edge relationships between taint objects. Further taint objects store edge relation attributes for children nodes, where the attributes pertains to the taint policy.

Currently the taint information is passed through to the visualizer in the form of a uniquely named text file. The text file is line-delimited and generated by the Analyzer component with the taint policy as the primary differentiating factor amongst taint graphs.

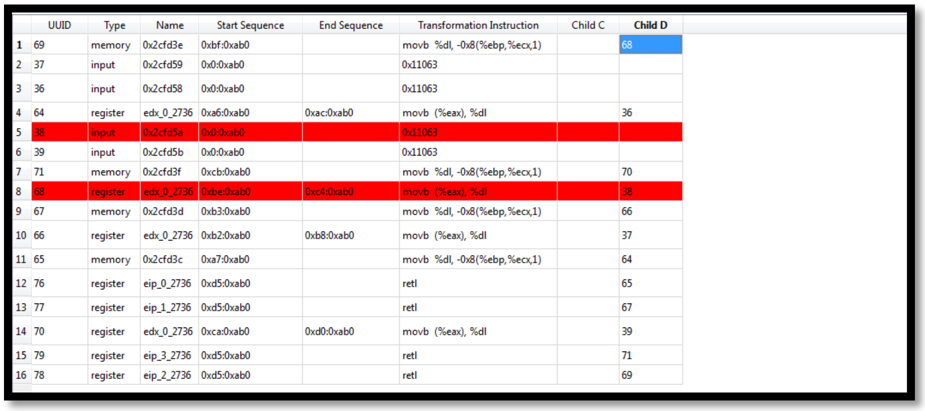
7.4 Architecture

The taint data is stored within a taint object data structure within memory that couples the taint attributes and edge relationships between taint objects. Further taint objects store edge relation attributes for children nodes, where the attributes pertains to the taint policy.

There are currently two concurrent graphing systems available in the visualizer, the IDA Pro grapher (WinGrapher) and a QT-based grapher. For the QT-based grapher, the layout algorithms currently rely on Scipy/Numpy in order to determine node and edge placement on a QT GraphScene.

7.5 TREE Visualizer Widget

TREE Visualizer exists as a QT Widget alongside the TREE Analyzer. As a whole, the plugin was developed utilizing IDAPython(Python interface for IDA Pro's APIs) and PySide(Python interface for PyQT). Figure 13 depicts the TREE Visualizer. The first component of the visualizer renders the taint data from the analyzer into a table to allow a user to sort on various characteristics. The figure also allows easy following the taint flow by highlighting all its descendants; for example, when a user selects node *69* 's child, child *68*, the direct child of *69*, and *38*, the descendant of child *68*, are both highlighted in red.



	UUID	Type	Name	Start Sequence	End Sequence	Transformation Instruction	Child C	Child D
1	69	memory	0x2cfd3e	0xbf:0xab0		movb %dl, -0x8(%ebp,%ecx,1)		68
2	37	input	0x2cfd59	0x0:0xab0		0x11063		
3	36	input	0x2cfd58	0x0:0xab0		0x11063		
4	64	register	edx_0_2736	0xa6:0xab0	0xac:0xab0	movb (%eax), %dl		36
5	38	input	0x2cfd5a	0x0:0xab0		0x11063		
6	39	input	0x2cfd5b	0x0:0xab0		0x11063		
7	71	memory	0x2cfd3f	0xcb:0xab0		movb %dl, -0x8(%ebp,%ecx,1)		70
8	68	register	edx_0_2736	0xbe:0xab0	0xc4:0xab0	movb (%eax), %dl		38
9	67	memory	0x2cfd3d	0xb3:0xab0		movb %dl, -0x8(%ebp,%ecx,1)		66
10	66	register	edx_0_2736	0xb2:0xab0	0xb8:0xab0	movb (%eax), %dl		37
11	65	memory	0x2cfd3c	0xa7:0xab0		movb %dl, -0x8(%ebp,%ecx,1)		64
12	76	register	eip_0_2736	0xd5:0xab0		retl		65
13	77	register	eip_1_2736	0xd5:0xab0		retl		67
14	70	register	edx_0_2736	0xca:0xab0	0xd0:0xab0	movb (%eax), %dl		39
15	79	register	eip_3_2736	0xd5:0xab0		retl		71
16	78	register	eip_2_2736	0xd5:0xab0		retl		69

Figure 13 – Taint data table

7.6 Interface

7.6.1 IDA Pro Grapher

IDA Pro uses WinGraph in generating visual graphs. An example IDA Graph, Figure 14, generated from BasicOV.exe, is depicted in figure 2. Taint nodes, as seen through the IDA Graph are currently uniquely distinguished by their name and color designation for nodes.

- Green - Input taint nodes
- Pink - Register taint nodes
- White - Memory taint nodes
- Red - Sink nodes

The visualizer's other focus point is the taint table. Children nodes are clickable and will show the sequence of taints through contextual highlighting of taint nodes within the table. The highlighting will allow a user to follow the flow of taints within a particular chain.

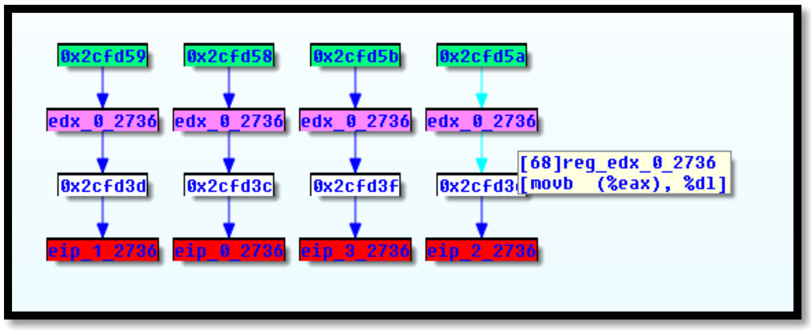


Figure 14 – BasicOV.exe Taint Graph

8 References

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

[2] Eagle, Chris. The IDA Pro Book: The Unofficial Guide to the World’s Most Popular Disassembler (2nd Edition). No Starch Press, 2011. Print.

[3] BIT team. TREE User Manual. http://code.google.com/p/tree-cbass/