Using Binary Code Instrumentation in Computer Security

Marius POPA[1], Sergiu CAPISIZU[2]
[1]Department of Economic Informatics and Cybernetics
Bucharest University of Economic Studies, Romania
[2]Bucharest Bar Association
marius.popa@ase.ro, capisizu@ew.ro

The paper approaches the low-level details of the code generated by compilers whose format permits outside actions. Binary code modifications are manually done when the internal format is known and understood, or automatically by certain tools developed to process the binary code. The binary code instrumentation goals may be various from security increasing and bug fixing to development of malicious software. The paper highlights the binary code instrumentation techniques by code injection to increase the security and reliability of a software application. Also, the paper offers examples for binary code formats understanding and how the binary code injection may be applied.
Keywords: *Binary Code, Code Instrumentation, Code Injection*

1 Binary Code Formats

The binary code is the output of the compiling process. It uses two binary digits, 0 and 1, to represent symbols or computer processor instructions as bit strings.

The binary code is directly executed by the computer processor or is interpreted by a specialized software component which translates the binary code into the format understood by the computer processor. Therefore, the binary code is classified into two groups:

- *Native code* – is generated by the compiler depending on the hardware features; the binary files containing native code are executed only by computer processor units (CPU) having the same features like the computer processor used to generate it; some significant advantages are execution speed and absence of other software components;
- *Intermediate code* – is generated by the compiler according to intermediate language specifications to be interpreted by a virtual machine; some examples of virtual machines are Java Virtual Machine (JVM) for Java platform and Common Language Runtime (CLR) for .NET framework; a big advantage is the binary code portability for different hardware and software platforms.

Some considerations related to binary code and file formats are presented in [1], [2], [4], [5], [6], [8] and [9].

The memory layout of native code file for a C program is depicted in figure 1 [11].

The user space is structured into the following segments:

- *Code segment* – contains executable instructions in binary format; it is a read-only segment and is shared among concurrent users;
- *Data segment* – contains static and global data initialized in the program code; each process has its own data segment; the data segment is not a executable one;
- *Block Started by Symbol (BSS) segment* – contains static and global data uninitialized in the program code; it is not an executable segment;
- *Stack segment* – is used to store local variables (declared inside the functions) and to pass parameters to functions; the stack addresses are allocated from higher memory to lower memory; it is managed by Stack Pointer (SP)/Extended Stack Pointer (ESP)/ Register Stack Pointer (RSP) register;
- *Heap segment* – is used to allocate memory at running time; heap allocation is managed by operating system; the

DOI: 10.12948/issn14531305/17.4.2013.05

heap addresses are managed by pointer variables.

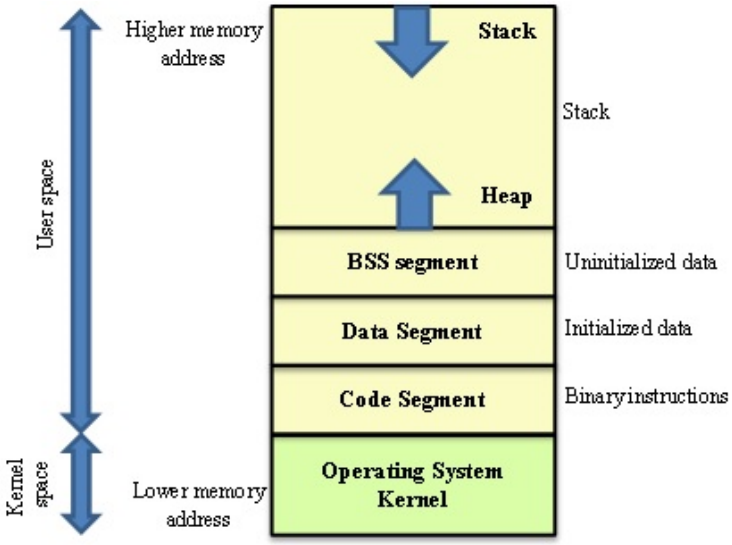

Fig. 1. Memory address space of a process

Segment registers are memory pointers placed inside the computer processor unit. They point to memory address where a data storage or code execution starts. The segment registers for x86 architecture are presented in Table 1.

Table 1. The segment registers for x86 computer architecture

Name	Content
CS – Code Segment	Pointer to the code location
SS – Stack Segment	Pointer to the stack location
DS – Data Segment	Pointer to the data location
ES – Extra Segment	Pointer to extra data
FS – F Segment	Pointer to extra data
GS – G Segment	Pointer to extra data

It considers the following C++ program source:

```
class Employee{
public:
    char* Name;
    int id;

    Employee(char* aName, int nr){
        this->Name = aName;
        this->id = nr;
        procData(aName, nr);
    }
    char* empName(){ return this->Name; }
    int empID(){ return this->id; }
    void procData(char* sName, int snr){ }
};

void main() {
      Employee e("Smith", 113);
      e.empID();
      e.empName();
}
```

DOI: 10.12948/issn14531305/17.4.2013.05

The native code built by Microsoft Visual Studio 2010, x64 version, for each method is presented in Table 2.

Table 2. Native code for `Employee` class

Method header	Memory address	Binary assembly instruction	Assembly instruction
`Employee(char*,int);`	00FF1440	55	push ebp
	00FF1441	8B EC	mov ebp,esp
	00FF1443	81 EC CC 00 00 00	sub esp,0CCh
	00FF1449	53	push ebx
	00FF144A	56	push esi
	00FF144B	57	push edi
	00FF144C	51	push ecx
	00FF144D	8D BD 34 FF FF FF	lea edi,[ebp-0CCh]
	00FF1453	B9 33 00 00 00	mov ecx,33h
	00FF1458	B8 CC CC CC CC	mov eax,0CCCCCCCCh
	00FF145D	F3 AB	rep stos dword ptr es:[edi]
	00FF145F	59	pop ecx
	00FF1460	89 4D F8	mov dword ptr [ebp-8],ecx
	00FF1463	8B 45 F8	mov eax,dword ptr [this]
	00FF1466	8B 4D 08	mov ecx,dword ptr [aName]
	00FF1469	89 08	mov dword ptr [eax],ecx
	00FF146B	8B 45 F8	mov eax,dword ptr [this]
	00FF146E	8B 4D 0C	mov ecx,dword ptr [nr]
	00FF1471	89 48 04	mov dword ptr [eax+4],ecx
	00FF1474	8B 45 0C	mov eax,dword ptr [nr]
	00FF1477	50	push eax
	00FF1478	8B 4D 08	mov ecx,dword ptr [aName]
	00FF147B	51	push ecx
	00FF147C	8B 4D F8	mov ecx,dword ptr [this]
	00FF147F	E8 11 FD FF FF	call Employee::procData (0FF1195h)
	00FF1484	8B 45 F8	mov eax,dword ptr [this]
	00FF1487	5F	pop edi
	00FF1488	5E	pop esi
	00FF1489	5B	pop ebx
	00FF148A	81 C4 CC 00 00 00	add esp,0CCh
	00FF1490	3B EC	cmp ebp,esp
	00FF1492	E8 9F FC FF FF	call @ILT+305(__RTC_CheckEsp) (0FF1136h)
	00FF1497	8B E5	mov esp,ebp
	00FF1499	5D	pop ebp
	00FF149A	C2 08 00	ret 8
`char* empName();`	00FF14C0	55	push ebp
	00FF14C1	8B EC	mov ebp,esp
	00FF14C3	81 EC CC 00 00 00	sub esp,0CCh
	00FF14C9	53	push ebx
	00FF14CA	56	push esi
	00FF14CB	57	push edi
	00FF14CC	51	push ecx
	00FF14CD	8D BD 34 FF FF FF	lea edi,[ebp-0CCh]
	00FF14D3	B9 33 00 00 00	mov ecx,33h
	00FF14D8	B8 CC CC CC CC	mov eax,0CCCCCCCCh
	00FF14DD	F3 AB	rep stos dword ptr es:[edi]
	00FF14DF	59	pop ecx
	00FF14E0	89 4D F8	mov dword ptr [ebp-8],ecx
	00FF14E3	8B 45 F8	mov eax,dword ptr [this]
	00FF14E6	8B 00	mov eax,dword ptr [eax]
	00FF14E8	5F	pop edi
	00FF14E9	5E	pop esi
	00FF14EA	5B	pop ebx
	00FF14EB	8B E5	mov esp,ebp
	00FF14ED	5D	pop ebp
	00FF14EE	C3	ret
`int empID();`	00FF1500	55	push ebp
	00FF1501	8B EC	mov ebp,esp

DOI: 10.12948/issn14531305/17.4.2013.05

	00FF1503	81 EC CC 00 00 00	sub esp,0CCh
	00FF1509	53	push ebx
	00FF150A	56	push esi
	00FF150B	57	push edi
	00FF150C	51	push ecx
	00FF150D	8D BD 34 FF FF FF	lea edi,[ebp-0CCh]
	00FF1513	B9 33 00 00 00	mov ecx,33h
	00FF1518	B8 CC CC CC CC	mov eax,0CCCCCCCCh
	00FF151D	F3 AB	rep stos dword ptr es:[edi]
	00FF151F	59	pop ecx
	00FF1520	89 4D F8	mov dword ptr [ebp-8],ecx
	00FF1523	8B 45 F8	mov eax,dword ptr [this]
	00FF1526	8B 40 04	mov eax,dword ptr [eax+4]
	00FF1529	5F	pop edi
	00FF152A	5E	pop esi
	00FF152B	5B	pop ebx
	00FF152C	8B E5	mov esp,ebp
	00FF152E	5D	pop ebp
	00FF152F	C3	ret
void procData (char*,int);	00FF1540	55	push ebp
	00FF1541	8B EC	mov ebp,esp
	00FF1543	81 EC CC 00 00 00	sub esp,0CCh
	00FF1549	53	push ebx
	00FF154A	56	push esi
	00FF154B	57	push edi
	00FF154C	51	push ecx
	00FF154D	8D BD 34 FF FF FF	lea edi,[ebp-0CCh]
	00FF1553	B9 33 00 00 00	mov ecx,33h
	00FF1558	B8 CC CC CC CC	mov eax,0CCCCCCCCh
	00FF155D	F3 AB	rep stos dword ptr es:[edi]
	00FF155F	59	pop ecx
	00FF1560	89 4D F8	mov dword ptr [ebp-8],ecx
	00FF1563	5F	pop edi
	00FF1564	5E	pop esi
	00FF1565	5B	pop ebx
	00FF1566	8B E5	mov esp,ebp
	00FF1568	5D	pop ebp
	00FF1569	C2 08 00	ret 8

The **`main`** function has the following native code content:

Table 3. Native code for `main` function

Function header	Memory address	Binary assembly instruction	Assembly instruction
void main();	00FF13A0	55	push ebp
	00FF13A1	8B EC	mov ebp,esp
	00FF13A3	81 EC D0 00 00 00	sub esp,0D0h
	00FF13A9	53	push ebx
	00FF13AA	56	push esi
	00FF13AB	57	push edi
	00FF13AC	8D BD 30 FF FF FF	lea edi,[ebp-0D0h]
	00FF13B2	B9 34 00 00 00	mov ecx,34h
	00FF13B7	B8 CC CC CC CC	mov eax,0CCCCCCCCh
	00FF13BC	F3 AB	rep stos dword ptr es:[edi]
	00FF13BE	6A 71	push 71h
	00FF13C0	68 3C 57 FF 00	push offset string "Smith" (0FF573Ch)
	00FF13C5	8D 4D F4	lea ecx,[e]
	00FF13C8	E8 14 FD FF FF	call Employee::Employee (0FF10E1h)
	00FF13CD	8D 4D F4	lea ecx,[e]
	00FF13D0	E8 35 FC FF FF	call Employee::empID (0FF100Ah)
	00FF13D5	8D 4D F4	lea ecx,[e]
	00FF13D8	E8 77 FD FF FF	call Employee:: empName (0FF1154h)
	00FF13DD	33 C0	xor eax,eax
	00FF13DF	52	push edx
	00FF13E0	8B CD	mov ecx,ebp
	00FF13E2	50	push eax

DOI: 10.12948/issn14531305/17.4.2013.05

	00FF13E3	8D 15 04 14 FF 00	lea edx, [(0FF1404h)]
	00FF13E9	E8 94 FC FF FF	call @ILT+125(@_RTC_CheckStackVars@8) (0FF1082h)
	00FF13EE	58	pop eax
	00FF13EF	5A	pop edx
	00FF13F0	5F	pop edi
	00FF13F1	5E	pop esi
	00FF13F2	5B	pop ebx
	00FF13F3	81 C4 D0 00 00 00	add esp,0D0h
	00FF13F9	3B EC	cmp ebp,esp
	00FF13FB	E8 36 FD FF FF	call @ILT+305(__RTC_CheckEsp) (0FF1136h)
	00FF1400	8B E5	mov esp,ebp
	00FF1402	5D	pop ebp
	00FF1403	C3	ret

Tables 2 and 3 contain numerical values for memory addresses and binary code in base-16 format. The assembly instructions and their binary codes are generated for x86 computer architecture.

The intermediate code provides the portability feature to a software application. A virtual machine interprets the operation codes from intermediate code and translates them into native code for the machine which runs the virtual machine. The operation codes are associated to intermediate instructions like the native code with the assembly instructions. The intermediate instructions are used within an intermediate language representing the lowest-level human-readable programming language.

For .NET Framework, the intermediate language is called Common Intermediate Language (CIL) and it is defined by Common Language Infrastructure (CLI). The CIL is object-oriented and stack-based. Unlike the CPU architecture, the stack-based architecture pushes data on a stack instead pulling data from registers.

The **`Employee`** class and **`main`** function defined above have the following content written in C# programming language:

```
class Employee
{
    public String Name;
    public int id;

    public Employee(String aName, int nr){
        this.Name = aName;
        this.id = nr;
        procData(aName, nr);
    }

    public String empName(){ return this.Name; }
    public int empID(){ return this.id; }
    public void procData(String sName, int snr) { }

    public static void Main() {
        Employee e = new Employee("Smith", 113);
        e.empID();
        e.empName();
    }
}
```

The intermediate code built by Microsoft Visual Studio 2010, x64 version, for each method is presented in Table 4.

DOI: 10.12948/issn14531305/17.4.2013.05

Table 4. C# .NET Intermediate code for `Employee` class

Method header	IL offset	Binary intermediate instruction	Intermediate instruction
Employee(String, int);	IL_0000	02	ldarg.0
	IL_0001	28 0A 00 00 11	call instance void [mscorlib]System. Object::.ctor()
	IL_0006	00	nop
	IL_0007	00	nop
	IL_0008	02	ldarg.0
	IL_0009	03	ldarg.1
	IL_000a	7D 04 00 00 01	stfld string EmployeeApp.Employee ::Name
	IL_000f	02	ldarg.0
	IL_0010	04	ldarg.2
	IL_0011	7D 04 00 00 02	stfld int32 EmployeeApp.Employee::id
	IL_0016	02	ldarg.0
	IL_0017	03	ldarg.1
	IL_0018	04	ldarg.2
	IL_0019	28 06 00 00 04	call instance void EmployeeApp. Employee::procData(string, int32)
	IL_001e	00	nop
	IL_001f	00	nop
	IL_0020	2A	ret
String empName();	IL_0000	00	nop
	IL_0001	02	ldarg.0
	IL_0002	7B 04 00 00 01	ldfld string EmployeeApp.Employee ::Name
	IL_0007	0A	stloc.0
	IL_0008	2B 00	br.s IL_000a
	IL_000a	06	ldloc.0
	IL_000b	2A	ret
int empID();	IL_0000	00	nop
	IL_0001	02	ldarg.0
	IL_0002	7B 04 00 00 02	ldfld int32 EmployeeApp.Employee::id
	IL_0007	0A	stloc.0
	IL_0008	2B 00	br.s IL_000a
	IL_000a	06	ldloc.0
	IL_000b	2A	ret
void procData (String,int);	IL_0000	00	nop
	IL_0001	2A	ret
static void Main();	IL_0000	00	nop
	IL_0001	72 70 00 00 01	ldstr "Smith"
	IL_0006	1F 71	ldc.i4.s 113
	IL_0008	73 06 00 00 01	newobj instance void EmployeeApp. Employee::.ctor(string, int32)
	IL_000d	0A	stloc.0
	IL_000e	06	ldloc.0
	IL_000f	6F 06 00 00 03	callvirt instance int32 EmployeeApp. Employee::empID()
	IL_0014	26	pop
	IL_0015	06	ldloc.0
	IL_0016	6F 06 00 00 02	callvirt instance string EmployeeApp. Employee::empName()
	IL_001b	26	pop
	IL_001c	2A	ret

The intermediate language (IL) offset represents the position of the current intermediate instruction relative to the beginning section where the method intermediate code is stored in the file translated by .NET virtual machine into native code. The intermediate code is organized in accordance with Portable Executable (PE) specifications for .NET applications.

The Java bytecode is the output of a Java compiler containing the binary content of the

DOI: 10.12948/issn14531305/17.4.2013.05

intermediate code. The bytecode files have a specific organization to be interpreted by JVM.

In Java programming language, the above example has the following content:

```
class Employee extends java.lang.Object {
public String Name;
public int id;

public Employee(String aName,int nr){
      this.Name = aName;
      this.id = nr;
      procData(aName, nr);
}
public String empName(){ return this.Name; }
public int empID(){ return this.id; }
public void procData(String sName,int snr) { }

public static void main(String[] args) {
        Employee e = new Employee("Smith", 113);
        e.empID();
        e.empName();
    }

}
```

The intermediate code built by Java compiler using Java Development Kit (JDK), 1.5.0 update 22 version, for each method is presented in table 5.

Table 5. Java bytecode for `Employee` class

Method header	Bytecode offset	Operation code	Bytecode instruction
public Employee(java.lang. String, int);	0	2A	aload_0
	1	B7 00 01	invokespecial #1; //Method java/lang/Object."<init>":()V
	4	2A	aload_0
	5	2B	aload_1
	6	B5 00 02	putfield #2; //Field Name:Ljava/lang/String;
	9	2A	aload_0
	10	1C	iload_2
	11	B5 00 03	putfield #3; //Field id:I
	14	2A	aload_0
	15	2B	aload_1
	16	1C	iload_2
	17	B6 00 04	invokevirtual #4; //Method procData:(Ljava/lang/String;I)V
	20	B1	return
public java.lang.String empName();	0	2A	aload_0
	1	B4 00 02	getfield #2; //Field Name:Ljava/lang/String;
	4	B0	areturn
public int empID();	0	2A	aload_0
	1	B4 00 03	getfield #3; //Field id:I
	4	AC	ireturn
public void procData(java.lang. String, int);	0	B1	return
public static void main(java.lang.Stri ng[]);	0	BB 00 05	new #5; //class Employee
	3	59	dup
	4	12 06	ldc #6; //String Smith
	6	10 71	bipush 113

DOI: 10.12948/issn14531305/17.4.2013.05

	8	B7 00 07	invokespecial #7; //Method "<init>":(Ljava/lang/String;I)V
	11	4C	astore_1
	12	2B	aload_1
	13	B6 00 08	invokevirtual #8; //Method empID:()I
	16	57	pop
	17	2B	aload_1
	18	B6 00 09	invokevirtual #9; //Method empName:()Ljava/lang/String;
	21	57	pop
	22	B1	return

JVM identifies the data structures where the operation codes of the intermediate instructions are stored in compiled Java file. JVM has a stack-based architecture used to translate the intermediate instructions to microcontroller or microprocessor native code. More details regarding the .NET and Java virtual machines are presented in [1], [2], [4], [5], [6], [8] and [9].

2 Binary Code Instrumentation for Malicious Software Detection

Code instrumentation is a technique that inserts code to analyze and modify the behavior a program [3], [10]. Depending on code representation, the code instrumentation is done for:

- *Source code* – text representation of the program;
- *Binary code* – binary encoded instructions of the program.

Code instrumentation techniques are applied for the following analysis levels:

- *Static analysis* – the program instrumentation occurs before the program running;
- *Dynamic analysis* – the program behavior is analyzed by collecting run-time information.

Static binary code instrumentation. It is method of analyzing the behavior of a binary application by modifying its object code or executable code before the program is run.

Static binary code instrumentation needs advanced knowledge about the executable formats like EXE and PE. The instrumentation techniques and analysis are more expensive than the dynamic instrumentation. Two important and particular issues are distinguished during static instrumentation, according to [7]:

- The binary application is statically disassembled; the disassembled code has not accurate information about the symbols and high level information regarding the symbols;
- Instrumented code requires more space than the original binary code; moving functions from the original binary code to a new location requires relocation information regarding the entry points provided by the compiler.

Static binary code instrumentation faces with the following challenges [7]:

- *Function boundaries and stack conventions* – in disassembled code, functions may have multiple entry and/or exit points; the function exit may be caused by a jump instead of return instruction; using the stack may follow unusual methods for Extended Stack Pointer (ESP) and Extended Base Pointer (EBP) register handling in x86-32bit computer architecture; therefore, there is a lack of information replaced by assumption information during the binary instrumentation;
- *Position-independent code* – there are multiple mechanisms for position code relative addressing; static binary instrumentation involves relocating of function without relocating of static data; this issue needs fixing the position code relative accesses or avoiding relocation of the binary sequences that accesses the relative addresses;
- *Scalability and modularity* – code instrumentation and analysis is done on modularity principle ensuring the scalability of the binary instrumentation to large programs, keeping the accuracy and speed of the process;

DOI: 10.12948/issn14531305/17.4.2013.05

- *Local variables* – may be accessed in different ways: using ESP as base pointer, using computer processor general register ECX by some functions like **`main`** in C language, using EBP to access the function parameter or in general purpose;
- *Actual parameters* – may be passed on the stack or via registers; they may be stored at stack-relative addresses or in the top of the stack when the binary code is optimized; the number of actual parameters are not established by a local examination of the function call;
- *Aliasing* – multiple pointer expressions may reference the same memory, increasing difficulty to identify the instructions that accesses that memory location; pointer analysis is very difficult without high-level information about variables, array sizes, types, and so forth;
- *Functions with side-effects* – some operations are performed by function itself instead of its caller, affecting the content of some variables and/or registers.

Binary instrumentation is a well-known technique used in computer security to analyze binary applications in exploit detection, sandboxing, malware analysis and so forth.

Dynamic binary code instrumentation. It is a method of analyzing the behavior of a binary application when is running. The analysis consists of code injection and examination of the effects on the binary code.

The benefits of the dynamic binary code instrumentation are highlighted in [10]:

- Recompiling or relinking is not needed;
- Code discovered at run-time;
- Generated code is handled dynamically;
- It is attached to running code.

A dynamic binary code instrumentation tool is Pin. Pin allows insertion of arbitrary code into executable when the program is running. Also, Pin can be attached to a running program [10].

Pin works like a "just in time" (JIT) compiler. Pin generates binary code that is executed. The code injection is done into generated binary code, the only code ever executed. Instrumentation is not done for instructions that are ever executed, being placed into conditional branches avoided by the execution flow of the program.

Dynamic binary code instrumentation done by Pin consists of two components [10]:

- A mechanism to establish the insertion points and what code to be inserted;
- The code to be executed at insertion points.

Pin tool implements the following instrumentation techniques for analyzed executable files [10]:

- *Instruction instrumentation* – includes the below possible operations:
  - Simple instruction counting – instruments a binary program to count the number of instructions executed; Pin tool inserts a call to count function before every instruction; the value of instruction count is saved in a file called **`inscount.out`** when the executable file finishes its running; the instruction count function is very simple, using a static integer variable and is passed as argument to a function called by Pin tools before every instruction encountered in the executable file; functions are defined in **`inscount0.cpp`** from Pin tool [10];

```
static UINT64 icount = 0;

VOID docount() { icount++; }
VOID Instruction(INS ins, VOID *v){
      INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_END);
}
```

DOI: 10.12948/issn14531305/17.4.2013.05

The **INS_InsertCall** function inserts a call to **docount** function relative to instruction **ins**. **IPOINT_BEFORE** is an enumerator always valid for all instructions and specifies the place of insertion. **IARG_END** is the argument list terminator for **INS_InsertCall** function.

- Instruction address tracing – instruments the memory addresses where the binary instructions are stored during execution of the program; Pin tool inserts a call to trace function before every instruction in the executable file; the function writes the content of Instruction Pointer (IP) register into an output file managed by a global variable called **trace** in **itrace.cpp** source file defined in Pin tool [10];

```
FILE * trace;

VOID printip(VOID *ip) {
    fprintf(trace, "%p\n", ip);
}
VOID Instruction(INS ins, VOID *v){
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)printip, IARG_INST_PTR,
IARG_END);
}
```

The **printip** function writes the content of IP register into the file managed by **trace** variable. The IP value is received by **ip** parameter and it has to be passed to **INS_InsertCall** function. The argument list of **INS_InsertCall** function is changed and the address of the instrumented instruction is passed by **IARG_INST_PTR** enumerator. Pin allows passing the content of different programming recipients by fields defined in the enumerator type called **IARG_TYPE**;

- Memory reference tracing – is a selective instrumentation technique, aiming a set of instructions from the binary code instrumented; for example, only instructions that read or write memory are considered; Pin tools inserts analysis function calls when each time when a memory operand is encountered during the execution flow of the binary program; the source code of this dynamic instrumentation technique is given in **pinatrace.cpp** source program and the results are saved into **pinatrace.out** file by the Pin tool [10];

```
FILE * trace;

VOID RecordMemRead(VOID * ip, VOID * addr){
    fprintf(trace,"%p: R %p\n", ip, addr);
}
VOID RecordMemWrite(VOID * ip, VOID * addr){
    fprintf(trace,"%p: W %p\n", ip, addr);
}
VOID Instruction(INS ins, VOID *v)
{
    UINT32 memOperands = INS_MemoryOperandCount(ins);

    for (UINT32 memOp = 0; memOp < memOperands; memOp++)
    {
        if (INS_MemoryOperandIsRead(ins, memOp))
```

DOI: 10.12948/issn14531305/17.4.2013.05

```
        {
            INS_InsertPredicatedCall(ins, IPOINT_BEFORE,
(AFUNPTR)RecordMemRead, IARG_INST_PTR, IARG_MEMORYOP_EA,
memOp, IARG_END);
        }
        if (INS_MemoryOperandIsWritten(ins, memOp))
        {
            INS_InsertPredicatedCall(ins, IPOINT_BEFORE,
(AFUNPTR)RecordMemWrite, IARG_INST_PTR, IARG_MEMORYOP_EA,
memOp, IARG_END);
        }
    }
}
```

The **RecordMemRead** function prints a read memory record to the output file. The **RecordMemWrite** function prints a write memory record to the output file. There is a single output file called **pinatrace.out**. The memory record has the layout:

(IP, memOpType, memAddr)

where:

- *IP* – Instruction Pointer register, containing the address of the instruction;
- *memOpType* – type of the memory operation; it takes ***R*** or ***W*** as values;
- *memAddr* – the memory address of the operand;

The **Instruction** function is called for each instruction, instrumenting the memory read and write operations. The number of memory operands is returned by **INS_MemoryOperandCount** function for each instruction **ins**. The memory operation type is checked by **INS_MemoryOperandIsRead** and **INS_MemoryOperandIsWritten** functions, requiring the **ins** instruction and **memOp** operand index. The **INS_InsertPredicatedCall** function avoids the predicated instructions with false predicate. The function call passes the instruction, the call place, the address of **RecordMemRead** or **RecordMemWrite** function, the instruction address, the effective address of the memory operand by **IARG_MEMORYOP_EA** enumerator, the operand index of the instruction, and the argument list terminator.

- *Image instrumentation* – aims loading and unloading of images. A trace file called **imageload.out** is created and contains trace messages related to loaded or unloading images. The source program containing the application for image instrumentation technique is **imageload.cpp** [10]. The needed functions are described below.

```
ofstream TraceFile;

VOID ImageLoad(IMG img, VOID *v){
    TraceFile << "Loading " << IMG_Name(img) << ", Image id = " <<
IMG_Id(img) << endl;
}

VOID ImageUnload(IMG img, VOID *v){
    TraceFile << "Unloading " << IMG_Name(img) << endl;
}
```

DOI: 10.12948/issn14531305/17.4.2013.05

```
IMG_AddInstrumentFunction(ImageLoad, 0);
IMG_AddUnloadFunction(ImageUnload, 0);
```

The **ImageLoad** function is called by Pin tool every time when an image is loaded. Also, the image can be instrumented. An image is a binary code sequence stored separately as the instrumented binary code. The **ImageUnload** function is called when a new image is unloaded. An image that is about to be unloaded cannot be instrumented. The **IMG_AddInstrumentFunction** function is called to register the **ImageLoad** callback to catch the loaded image. The **IMG_AddUnloadFunction** function call passes the image that is about to be unloaded and the value of **ImageUnload** function. The unloaded images cannot be instrumented, so the **ImageUnload** is not an instrumentation function.

- *Trace instrumentation* – is a more efficient instruction counting. The counter is incremented per basic block instead of each instruction. In [10], trace instrumentation is implemented in **inscount1.cpp** application.

```
static UINT64 icount = 0;

VOID docount(UINT32 c) {
    icount += c;
}
VOID Trace(TRACE trace, VOID *v){
    for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl);
bbl = BBL_Next(bbl)) {
        BBL_InsertCall(bbl, IPOINT_BEFORE, (AFUNPTR)docount,
IARG_UINT32, BBL_NumIns(bbl), IARG_END);
    }
}
```

The **docount** function is called before every basic block. The **trace** parameter is an object assigned to a sequence of instructions with a single entrance and multiple exits. The **for** instruction parses the basic block set of the binary application and inserts a call to **docount** function before every basic block by **BBL_InsertCall** function. The **BBL_InsertCall** function call passes the basic block to be instrumented, the call place, the instrumentation function, the argument of **docount** function, number of instructions within the **bbl** basic block and argument list terminator.

- *Routine instrumentation* – is made in [10] by number of calls for each procedures and the number of instructions executed in each procedure. The instrumentation report is printed to **proccount.out** file, and the source file of the application is **proccount.cpp**.

```
ofstream outFile;

typedef struct RtnCount {
    string _name;
    string _image;
    ADDRINT _address;
    RTN _rtn;
    UINT64 _rtnCount;
    UINT64 _icount;
    struct RtnCount * _next;
```

DOI: 10.12948/issn14531305/17.4.2013.05

```
} RTN_COUNT;

RTN_COUNT * RtnList = 0;

VOID docount(UINT64 * counter) {
    (*counter)++;
}

const char * StripPath(const char * path) {
    const char * file = strrchr(path,'/');
    if (file)
        return file+1;
    else
        return path;
}

VOID Routine(RTN rtn, VOID *v) {
    RTN_COUNT * rc = new RTN_COUNT;

    rc->_name = RTN_Name(rtn);
    rc->_image = StripPath(IMG_Name(SEC_Img(RTN_Sec(rtn))).c_str());
    rc->_address = RTN_Address(rtn);
    rc->_icount = 0;
    rc->_rtnCount = 0;
    rc->_next = RtnList;
    RtnList = rc;
    RTN_Open(rtn);

    RTN_InsertCall(rtn, IPOINT_BEFORE, (AFUNPTR)docount, IARG_PTR,
&(rc->_rtnCount), IARG_END);

    for (INS ins = RTN_InsHead(rtn); INS_Valid(ins); ins = INS_Next(ins)){
        INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount,
IARG_PTR, &(rc->_icount), IARG_END);
    }
    RTN_Close(rtn);
}
```

The **Routine** function is called every time when a new routine is executed. The **rc** local variable is a simple linked list to store information related to current routine. The current running routine is added to global list of routine managed by **RtnList** global variable. The call count of the current routine is incremented by **RTN_InsertCall** function call. The instruction counter for each routine is also stored by the simple linked list. The incrementing of instruction counter is made by **INS_InsertCall** function call. The routines are managed by routine objects and a certain routine is instrumented by referring the object name.

Binary code instrumentation provides information about the binary applications and helps to improve the correctness and speed of the developed programs. Also, the program behavior checking provides valuable information about third-party application, identifying the suspicious behaviors of the programs from unsecure sources.

3 Methods and Techniques of Binary Code Injection

Binary code injection inserts a binary sequence into a binary computer program to change the intended course of execution.

A way to change the course of execution in native executable files is altering of IP register. IP is a special-purpose register that stores the memory address of the next instruction to be executed. It cannot be

DOI: 10.12948/issn14531305/17.4.2013.05

programmatically accessed and its content is altered by instructions such as **`jmp`**, **`call`** and **`ret`**. These instructions permit the access to old IP values automatically pushed onto stack.

The stack content for x86-16bit computer architecture when a function is called is depicted in Figure 2.

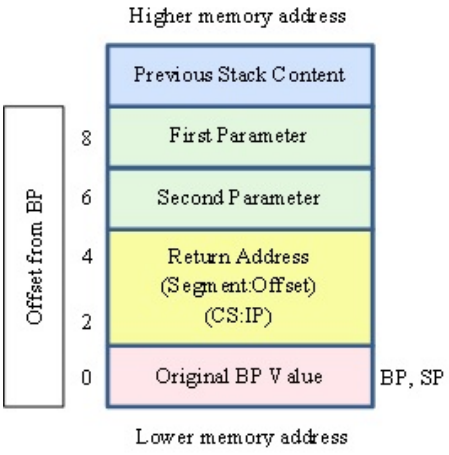

Fig. 2. Accessing stack content in a far function

The above stack content is available after a far function call with two arguments. All local variables are allocated on stack at lower memory addresses, and the return address to caller has the offsets from the current BP as 2 bytes for IP register restore and 4 bytes for CS register restore.

The IP content is restored when the **`ret`** instruction is executed. All these issues together with a stack content management may cause the change the course of execution.

To get the current value of the IP register, a dummy function is defined and its call is followed by extracting the value from the top of the stack. Therefore, the stack portion where the return address is stored may be overwritten by the extracted value of IP, and the program execution is altered.

For x86-16bit computer architecture, the register size is 16 bits. Also, the computer word has 16 bits length. For x86-32bit computer architecture, the register size and computer word are extended to 32 bits. The IP, BP and SP registers are included in the extended versions called EIP, EBP and ESP registers.

An example for IP content handling is presented below. The binary code of the sequence is injected into the binary code of the **`Employee`** constructor method presented in chapter 1.

Table 6. Native code injection

Method header	Memory address	Binary assembly instruction	Assembly instruction
`Employee(char*,int);`	`;previous addresses`	`;previous binary instructions`	`;previou asm instructions`
	`00FF146B`	`8B 45 F8`	`mov eax,dword ptr [this]`
	`00FF146E`	`8B 4D 0C`	`mov ecx,dword ptr [nr]`
	`00FF1471`	`89 48 04`	`mov dword ptr [eax+4],ecx`
		`;injected binary instructions`	`;injected asm instructions`
	`00FF1474`	`E8 00 00 00 00`	`call dummy (0FF1479h)`
	`00FF1479`	`59`	`pop ecx`
	`00FF147A`	`89 4D 08`	`mov dword ptr [aName],ecx`
		`;old binary sequence`	`;old asm sequence`
	`00FF147D`	`8B 45 0C`	`mov eax,dword ptr [nr]`
	`00FF1480`	`50`	`push eax`
	`00FF1481`	`8B 4D 08`	`mov ecx,dword ptr [aName]`
	`00FF1484`	`51`	`push ecx`
	`00FF14A3`	`C2 08 00`	`ret 8`

DOI: 10.12948/issn14531305/17.4.2013.05

The effect of the injected **`Employee`** code in constructor method is a wrong **`aName`** string parameter passed to **`procData`** method. The IP content popped by ECX register is the address of the **`pop ecx`** instruction. A disastrous effect is produced when the restored IP register contains a malicious code address.

Also, elaborated stack buffer overflow attacks can be carried out by redirecting the execution flow to the shellcode that overwrites the vulnerable buffer of the target host.

The bytecode injection consists in changing the existing bytecode to redirect the execution flow built by virtual machine. For Java programming language, that means modification of the compiled file stored by the class file and interpreted by the Java Virtual Machine. That issue is available for programs with malicious intent. However, there are situations when code injection techniques are used for bytecode instrumentation of the third-party libraries which the source code is not available or a debugger or a profiler cannot be used. Also, the bytecode injection is used in performance monitoring of the application [12].

The bytecode injection allows modification of third-party bytecode without the source code availability. The injection is asked by existence of some bugs or limitations of the third-party software to see the behavior change and possible solutions to pass over the software problems. The following considerations are taken into account by [12] as good reasons to apply injection techniques:

- The exception generated by a method during a data set processing provides no information about what data element has caused it;
- Collecting performance statistics;
- The failure of a data batch processing when a data set is send to a database by Java Database Connectivity (JDBC).

Advantages of Java bytecode injection are presented in [12]:

- Modification of the binary code when the source code is not available;
- Collecting run-time information when tools like debuggers and profilers cannot be used;
- Modularity of the injected bytecode;
- Keeping the original bytecode by using tools for bytecode injection;
- Doing injection at compile-time when the bytecode is built or run-time when the target classes are loaded by the JVM.

The Java bytecode format is presented in [1], [2], [4], [5], [6], [8] and [9]. Depending on level of abstraction, the injection tools aim [12]:

- Direct bytecode manipulation (ASM tool)– Java bytecode understanding is needed because the level of abstraction is very low; the developer has to work with operation codes, the operand stack and bytecode instructions;
- Intermediate level (Javassist tool) – the code is given in strings, the classfile structure has a level of abstraction;
- Advices in Java (AspectJ tool) – the code to be injected has a syntax-checked format and it is compiled.

The injection techniques are classified by [12] into the following categories:

- *Manual injection* – the developer knows the place where the code is injected; to do that, the developer must know the classfile format;
- *Primitive pointcuts* – pointcut is an expression telling the place where a particular bytecode must be injected; this injection techniques has limitation about the place: a particular method, all public methods of a class and so forth;
- *Pattern matching pointcut expressions* – match the target bytecode based on a number of criteria.

The bytecode injection time may be [12]:

- *Manually at run-time* – the bytecode asks for injected bytecode;
- *Load-time* – bytecode injection is performed when the target bytecode is loaded by the JVM;
- *Build-time* – the bytecode is modified by injection before packaging and deploying the software application.

DOI: 10.12948/issn14531305/17.4.2013.05

According to [1], [2], [4], [5], [6], [8] and [9] where the Java classfile structure is explained, injection of a new method into classfile involves the structure information modifications inside the classfile in addition to the method bytecode. Modifications are manually done in the classfile or implemented in libraries as injection tool. The operations needed to keep a viable classfile to be correctly interpreted by the JVM are:

- Addition of the method in the Constant Pool – write the information as a constant pool entry:
  - Adding UTF8 method name to constant pool;
  - Adding UTF8 descriptor index to constant pool;
  - Adding method name type to constant pool;
  - Adding method type to constant pool;
- Injection of the method bytecode – inserting the method binary code into the classfile:
  - Insertion of the method bytecode;
  - Write the bytecode image;
  - Adjusting the offsets: code length, maximum stack, exception table, code attributes, and attribute length.

The new content of the classfile has to meet the structural constraints to pass the verification proceeded by JVM in order to interpret the bytecode correctly.

4 Conclusions

Binary code instrumentation offers the tools needed to increase the security and reliability of a software application. Therefore, the developers are able to understand how an exploit is created to pass the security mechanisms of the application in order to build defense techniques.

Also, the binary code instrumentation may be used reverse engineering of malwares in order to detect them and implement protection techniques.

The system security depends on the safety of running binary code, operating system kernel internals, linker and loader internals that have to be known and understood by software developers.

References

[1] C. Boja and M. Doinea, "Security Assessment of Web Based Distributed Applications", *Informatica Economică*, vol. 14, no. 1, 2010, pp. 152 – 162

[2] A Danehkar, "Inject your code to a Portable Executable File", 27 December 2005, http://www.codeproject.com [Nov. 15, 2013]

[3] F. Falcon and N. Riva, "Dynamic Binary Instrumentation Frameworks: I Know You're there Spying on Me", Core Security, June 2012

[4] R. Paleari, "Static disassembly and analysis of malicious code", 5 July 2007, http://roberto.greyhats.it/talks.html [Nov. 15, 2013]

[5] M. Pietrek, "An In-Depth Look into the Win32 Portable Executable File Format", *MSDN Magazine*, http://msdn.microsoft.com /en-us/magazine/cc301805.aspx [Nov. 15, 2013]

[6] M. Popa, "Binary Code Disassembly for Reverse Engineering", *Journal of Mobile, Embedded and Distributed Systems*, vol. 4, no. 4, 2012, pp. 233 – 248,

[7] P. Saxena, R. Sekar and V. Puranik, "Efficient Fine-Grained Binary Instrumentation with Applications to Taint-Tracking", http://seclab.cs.stonybrook.edu/seclab/pubs/cgo065-saxena.pdf [Nov. 15, 2013]

[8] C. Toma, "Sample Development on Java Smart-Card Electronic Wallet Application", *Journal of Mobile, Embedded and Distributed Systems*, vol. 1, no. 2, 2009, pp. 60 – 80

[9] G. Vigna, "Static Disassembly and Code Analysis, Malware Detection. Advances in Information Security", Springer, Heidelberg, vol. 35, 2007, pp. 19 – 42

[10] Pin 2.13 User Guide, Intel Developer Zone, http://software.intel.com/sites/landingpage/pintool/docs/61206/Pin/html/ [Nov. 15, 2013]

[11] http://rohanpuri.blogspot.ro/

DOI: 10.12948/issn14531305/17.4.2013.05

2010_08_01_archive.html [Nov. 15, 2013]
[12] http://theholyjava.wordpress.com/2011/09/07/practical-introduction-into-code-injection-with-aspectj-javassist-and-java-proxy/ [Nov. 15, 2013]

Marius POPA has graduated the Faculty of Cybernetics, Statistics and Economic Informatics in 2002. He holds a PhD diploma in Economic Cybernetics and Statistics. He joined the staff of Academy of Economic Studies, teaching assistant in 2002. Currently, he is Associate Professor in Economic Informatics field and branches within Department of Economic Informatics and Cybernetics at Faculty of Cybernetics, Statistics and Economic Informatics from Bucharest University of Economic Studies. He is the author and co-author of 9 books and over 140 articles in journals and proceedings of national and international conferences, symposiums, workshops in the fields of data quality, software quality, informatics security, collaborative information systems, IT project management, software engineering.

Sergiu CAPISIZU has graduated the Faculty of Cybernetics, Statistics and Economic Informatics in 1997 and National University of Defense in 2005. He holds a PhD diploma in Economic Cybernetics and Statistics, having the title Models and techniques to perform the economic information audit. He is co-author of books and articles in information audit and ICT fields. Also, he has published articles in proceedings of national and international conferences, symposiums, workshops in the fields of data quality, software quality, information audit and juridical aspects in ICT field. He is evaluator of ANEVAR association.

DOI: 10.12948/issn14531305/17.4.2013.05