CS266 Software Reverse Engineering (SRE)
Applying Anti-Reversing Techniques to Machine Code

Teodoro (Ted) Cipresso, teodoro.cipresso@sjsu.edu
Department of Computer Science
San José State University
Spring 2015

The information in this presentation is taken from the thesis “Software reverse engineering education” available at http://scholarworks.sjsu.edu/etd_theses/3734/ where all citations can be found.
Applying Anti-Reversing Techniques to Machine Code

Introduction, Motivation, and Considerations

- Extreme care must be taken when applying anti-reversing techniques because most techniques ultimately change the machine code.

- In the end, if a program does not run correctly, measuring how efficient or difficult to reverse engineer it is becomes meaningless [18].

- Anti-reversing transformations performed on source code make a program more difficult to understand in both source and executable formats.
  - These transforms can expose compiler bugs because the program no longer looks like something a human would write.
  - [18] states “any compiler is going to have at least some pathological programs which it will not compile correctly.”
Applying Anti-Reversing Techniques to Machine Code
Introduction, Motivation, and Considerations

- Compiler failures on “pathological” programs occur because compiler test cases are most often coded by people.

- Test cases are not typically generated by some sophisticated tool that knows how to try every fringe case and surface every bug.

- Therefore we should not be surprised if some compilers have difficulty with obfuscated source code.

- We now investigate the technique *Eliminating Symbolic Information* as it applies to machine code.

  - We previously looked at this technique in *Applying Anti-Reversing Techniques to Java Bytecode*. 
Applying Anti-Reversing Techniques to Machine Code

Eliminating Symbolic Information in Machine Code

- **Eliminating Symbolic Information** calls for the removal of any meaningful symbolic information in machine code that is not important to the execution of the program, but serves to ease debugging or reuse of it by another program.

  - For example, if a Windows program references functions (methods) exported by one or more libraries (DLLs), the names of those methods will appear in the .idata (import data) section of the program binary.

- In production versions of a program, the machine code doesn't directly contain any symbolic information from the original source code. In the executable:
  - Method names, variable names (etc..), and line numbers are all absent.
  - Only the machine instructions produced directly or indirectly by the compiler [9] are present.
Applying Anti-Reversing Techniques to Machine Code
Eliminating Symbolic Information in Machine Code

- This lack of information about the connection between the machine instructions and HLL source is unacceptable for purposes of debugging.
  - Therefore most modern compilers, like GCC, include an option to include debugging information into an executable.
  - The included debugging information allows a debugger to map one or more machine instructions back to a particular HLL statement [9].

- To demonstrate symbolic information that is inserted into machine code to enable debugging of an application:
  - `Calculator.cpp` was compiled using the GNU C++ compiler (g++) with options to include debugging information and to generate assembly source instead of an executable (machine code).
Applying Anti-Reversing Techniques to Machine Code
Eliminating Symbolic Information in Machine Code

```
using namespace std;

long doAdd(int op1, int op2) { return op1 + op2; }
long doSub(int op1, int op2) { return op1 - op2; }
long doMul(int op1, int op2) { return op1 * op2; }

int main(int argc, char *argv[])
{
    string input; int op1, op2; char fnc; long res;
    cout << "Enter integer 1: ";
    getline(cin, input); op1 = atoi(input.c_str());
    cout << "Enter integer 2: ";
    getline(cin, input); op2 = atoi(input.c_str());
    cout << "Enter function [+][-][*]: ";
    getline(cin, input); fnc = input.at(0);
    switch (fnc)
    {
    case '+':
        res = doAdd(op1, op2); break;
    case '-':
        res = doSub(op1, op2); break;
    case '*':
        res = doMul(op1, op2); break;
    }
    cout << "Result: " << res << endl;
    return 0;
}
```

Applying Anti-Reversing Techniques to Machine Code
Eliminating Symbolic Information in Machine Code

- The GNU compiler stores debug information in the *symbol tables* (.stabs) section of the Windows PE header so that it will be loaded into memory as part of the program image.

- The generated assembly language files `CalculatorDebug.s` on the next slide shows some of the debugging information inserted by GCC.
  
  - While this information is not a replacement for the original source code it does provide some helpful information to a reverser.
  
  - The GNU Project Debugger (GDB) is a source-level debugger and therefore must access the HLL source file to make use of the debugging information.
Applying Anti-Reversing Techniques to Machine Code
Eliminating Symbolic Information in Machine Code

581 \texttt{.lcomm \_ZstL8\_jinit,1,1}
582 \texttt{.stabs \_Z5doAddii:F(0,19),36,0,0,\_Z5doAddii}
583 \texttt{.stabs \texttt{"op1:p(0,12),160,0,0,16}}
584 \texttt{.stabs \texttt{"op2:p(0,12),160,0,0,24}}
585 \texttt{.globl _Z5doAddii}
586 \texttt{.def \_Z5doAddii; \_scl 2; \_type}
587 \texttt{.seh\_proc _Z5doAddii}
588 \texttt{\_Z5doAddii:}
589 \texttt{.stbn 68,0,7,\_LMO-.LFBB1}
590 \texttt{\_LMO:}
591 \texttt{\_LFBB1:}
592 \texttt{\_LFB978:}
593 \texttt{\textbf{pushq} \%rbp}
594 \texttt{\_seh\_pushreg \%rbp}
595 \texttt{\textbf{movq} \%rsp, \%rbp}
596 \texttt{\_seh\_setframe \%rbp, 0}
597 \texttt{\_seh\_endprologue}
598 \texttt{\textbf{movl} \%ecx, 16(\%rbp)}
599 \texttt{\textbf{movl} \%edx, 24(\%rbp)}
600 \texttt{.stbn 68,0,7,\_LM1-.LFBB1}
601 \texttt{\_LM1:}
602 \texttt{\textbf{movl} 24(\%rbp), \%eax}
603 \texttt{\textbf{movl} 16(\%rbp), \%edx}
604 \texttt{\textbf{addl} \%edx, \%eax}
605 \texttt{\textbf{cld}}
606 \texttt{\textbf{popq} \%rbp}
607 \texttt{\textbf{ret}}
608 \texttt{\_seh\_endproc}
Applying Anti-Reversing Techniques to Machine Code
Eliminating Symbolic Information in Machine Code

- Debugging information can give plenty of hints to a reverse engineer, such as the count and type of parameters one must pass to a given method.

- An obvious recommendation is to ensure the code is not compiled with debugging information before shipment to end-users.

- The hunt for symbolic information doesn't end with information embedded by debuggers; actually it rarely begins there.

- Eliminating symbolic information in machine code is difficult, therefore we’ll manually implement some familiar techniques to obfuscate the information.

- First, we will take a detour to look at obfuscation of source code and what use cases it might address.
Applying Anti-Reversing Techniques to Machine Code
Source Code Obfuscation

- **Obfuscating the Program** calls for performing transformations to the source or machine code that would render either code extremely difficult to understand but functionally equivalent to the original.

- When delivering software to customers, they may require the source code so that the product can be compiled using in-house build and audit procedures.

- In the likely event that the source code contains intellectual property, it can be obfuscated without changing the resultant machine code.

- To demonstrate source code obfuscation, COBF [23], a free C/C++ source code obfuscator was configured and given Calculator.cpp.
  - **Stunnix C/C++** is a commercial source code obfuscator.
#include "cobf.h"

#define l2 using
#define lw namespace
#define lz std
#define lg long
#define lc int
#define lf return
#define lu main
#define ln char
#define lo string
#define lh cout
#define li getline
#define lo atoi

#define lr(la,lc,lb){lf la+lb;}lg lp(lc,la,lc,lb){lf la-lb;}lg
#define lr(la,lc,lb){lf la*lb;}lc lu(lc,lla,ln*ll[ll]){ll ld;lc la,lb;ln 1k;
#define le lh="x45\x6e\x74\x65\x72\x20\x69\x6e\x74\x65\x67\x65\x72\x20"
#define le lh="x31\x3a\x20";li(ll,lk);la=lo(ll.ls()); lh="x45\x6e\x74\x65\x72\x20"
#define le lh="x69\x6e\x74\x65\x72\x20\x32\x3a\x20";li(ll,lk);lk=lo(ll.ls();
#define le lh="x45\x6e\x74\x65\x72\x20\x20\x75\x6e\x63\x74\x69\x66\x6f\x20"
#define le lh="x5b\x2b\x7c\x2d\x7c\x2a\x5d\x3a\x20";li(lj,lk);lk=ld.lx(0);lv(lk){
#define le lh="x52\x65\x73\x75\x6c\x74\x3a\x20"ll;lm'+'=lr(la,lb);ll;lm'-'=lp(la,lb);ll;lm'*'=lq(la,lb);ll;}

<<"\x52\x65\x73\x75\x6c\x74\x3a\x20"<<le<<ly;lf 0;"
Applying Anti-Reversing Techniques to Machine Code

Basic Obfuscation of Machine Code

- [19] states “Obfuscation of Java bytecode is possible for the same reasons that decompiling is possible: Java bytecode is standardized and well documented.”

- Machine code is not standardized; instruction sets, formats, and program image layouts vary depending on the target platform architecture.

- Tools to assist with obfuscating machine code are much more challenging to implement and expensive to acquire. *(I have not found any free tools).*

- EXECryptor is an industrial-strength machine code obfuscator.
  
  - When applied to the machine code for the Password Vault application, EXECryptor rendered it extremely difficult to understand.

    - EXECryptor fails to start in Windows 8.1 due to an anti-debug error.
Applying Anti-Reversing Techniques to Machine Code

Basic Obfuscation of Machine Code

- Direct machine code obfuscations can be hard to analyze or follow, so we'll perform obfuscations at the source code level and observe differences in the assembly code generated by the GNU C/C++ compiler.

- Success is achieved when an obfuscated program has the same functionality as the original, but is more difficult to understand during live or static analysis.

- There are no standards for code obfuscation, but it's relatively important to ensure that the obfuscations applied to a program are not easily undone because deobfuscation tools can be used to eliminate easily identified obfuscations [5].

- We now look at the source and disassembly for `VerifyPassword.cpp`, a simple C++ program that contains a simple if-test for a password.
  - We then embed a simple cipher to protect sensitive constants.
Applying Anti-Reversing Techniques to Machine Code

Basic Obfuscation of Machine Code

```cpp
1 using namespace std;
2
3 int main(int argc, char *argv[])
4 {
5     const char *password = "jupiter";
6     string specified;
7     cout << "Enter password: ";
8     getline(cin, specified);
9     if (specified.compare(password) == 0) {
10        cout << "[OK] Access granted."
11        << endl;
12     } else {
13        cout << "[Error] Access denied."
14        << endl;
15     }
16 }
```
.text section

.rdata section
Applying Anti-Reversing Techniques to Machine Code
Basic Obfuscation of Machine Code

```cpp
#include "substitutionciphertext.h"

using namespace std;

int main(int argc, char *argv[]) {
    const char *password = "77827D2E81727F";
    const char *enter_password = "527B81727F2D7D6E8080847C7F71472D";
    const char *password_ok = "685C586A2D4E7070728082D747F6E7B8172713B";
    const char *password_bad = "68527F7F7C7F6A2D4E7070728082D71727B7672713B";
    SubstitutionCiphertext cipher;
    string specified;
    cout << cipher.decryptFromHex(enter_password);
    getline(cin, specified);
    if (specified.compare(cipher.decryptFromHex(password)) == 0) {
        cout << cipher.decryptFromHex(password_ok) << endl;
    } else {
        cout << cipher.decryptFromHex(password_bad) << endl;
    }
}
```
End