Ihimu Ukpo

SID: iuu1

E-mail: iuu1@columbia.edu

COMS E6156, Spring 2016

Final Report

Deliverable Detail

All of Proteus' source code has been placed in a publicly available GitHub repo: https://github.com/iukpo/Proteus. There is no special access required; anyone may browse the source at the URL provided, or do a git clone to make a local copy of the repository.

External Software

Proteus is written in Java and requires the following open source packages:

- Apache Commons FileUtils
- Apache Commons Lang 3.4

Proteus has an assistant program that is written in C/C++ that is dependent on a modified version of Homomorphic Encryption Library (HELib).

Limitations in Implementation

Due to time constraints, Proteus has the following limitations:

- Proteus can only build programs with Makefiles.
- Anti-debugging protection is not available for Java programs, as no homomorphic encryption library written in pure Java is available.
- Inward jumps are written in x86 assembler only.
- Only equality comparisons involving integers are supported for anti-debugging protection.
- Only byte and integer types are supported for Java anti-disassembly code.
- Only char, byte, and integer types are supported for C/C++ anti-debugging/anti-disassembly protection.
- I couldn't come up with a way to safely store private keys used in the homomorphic functions. I ran out of time, but plan to fix this in a future release.

Lessons Learned

While opaque predicates provide cheap (easily implemented) and resilient protection against disassembly, they are not as effective alone in the Java language. Without other forms of protection, decompiled classes can map directly to their source code (see Figures one and two). While opaque predicates can still provide protection under such circumstances, the protection is substantially diminished, as the attacker can easily identify the opaque predicate and match it against any known opaque predicates. Doing so would allow the attacker to determine which part of the predicate to ignore. When applied to Java source code, opaque predicates should therefore be used in conjunction with other anti-disassembly techniques such as name obfuscation and class encryption. I argue that they are still useful because they can force an attacker to guess which code will be executed if an attacker only has access to a disassembly or decompilation.

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class AsciiDraw {

    private static byte[] output;

    public static void initImage(String imgFilename)
    {
        try
        {
            //convert file into array of bytes
            File file = new File(imgFilename);
            output=new byte[(int) file.length()];
            FileInputStream fileInputStream = new FileInputStream(file);
            fileInputStream.read(output);
            fileInputStream.close();
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }

    public static void decryptAndDisplay()
    {
        int i=0;
        char theKey='*';

        for (i = 0; i < output.length; i++)
        {
            /*<antidisasm><platform>java</platform><input><inputname>output[i]</inputname><inputdataty
                output[i]=(byte)(output[i]+theKey);
        }

        String s = new String(output);

        System.out.println(s);
    }
```

Figure 1: Screenshot of unprotected Java source code. The second comment is an anti-disassembly mark.

While the cost of implementing opaque predicates for anti-disassembly code is low, the same cannot be said here of anti-debugging code via homomorphic encryption. Although debugging homomorphic code is useless in the pursuit of locating sensitive data in memory, there are several downsides to implementing homomorphic encryption.

Unlike anti-disassembly code, which can be implemented in Proteus without any action on the author's part aside placing marks in the code telling Proteus where to place anti-disassembly protection,

anti-debugging protection provided by homomorphic encryption requires more than just placing anti-debugging marks. It requires the software author to include a header file containing necessary function prototypes and link against additional libraries (a static library built from HELib_Proteus-FHE.a-as well as the GNU Multiple Precision Arithmetic Library). Without optimization, the resulting binary can swell up considerably in size. For example, the AsciiDraw sample program written in C/C++ yields a binary of size of 178kb. The same program with homomorphic encryption is 28MB in size.

```
import java.io.BufferedReader;

public class AsciiDraw
{
  private static byte[] output;

  public static void initImage(String imgFilename)
  {
    try
    {
      File file = new File(imgFilename);
      output = new byte[(int)file.length()];
      FileInputStream fileInputStream = new FileInputStream(file);
      fileInputStream.read(output);
      fileInputStream.close();
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
  }

  public static void decryptAndDisplay()
  {
    int i = 0;
    char theKey = '*';
    for (i = 0; i < output.length; i++) {
      output[i] = ((byte)(output[i] + theKey));
    }
    String s = new String(output);

    System.out.println(s);
  }
```

Figure 2: Disassembly of Java class produced from source in figure one. Note that the disassembly is nearly identical to the source code.

Another penalty in including homomorphic encryption is that the target program will now require several files to run: a public key (for encrypting data), a private key (for decrypting data where necessary), and the files containing the encrypted sensitive data. This introduces run-time requirements to the program being protected: without the keys and the encrypted data, the program will not run. The current handling of the keys further hurts the situation: while the keys are given randomly generated filenames, they are stored in the same location as the program. Instead of trying to use a debugger to ascertain sensitive data, an attacker could simply use trial and error to determine which file corresponds

to the keys and which files correspond to the encrypted data. Future plans for Proteus include fixing this problem.

The last penalty in implementing homomorphic encryption is time. Homomorphic encryption is very slow. While integer comparisons happen instantaneously, the same process involving homomorphic encryption takes several seconds to execute. If one plans to use homomorphic encryption at all, it should be in areas of code where the time penalty does not have a significant effect on performance.