An API For API Hookers
Taking A Closer At Malware

Stuart Maclean

Applied Physics Laboratory
University of Washington

Open Source Digital Forensics Conference, 2013
Outline

1. Motivation
2. API Hooking Obstacles
3. A Parser For Windows Header Files
4. An API For Autogenerating Hook Functions
5. Collecting The Hooked Call Information
6. Conclusion
Motivation

Want to answer the question: what does randomBinary.exe do?

- We could stare at the assembler and/or run it.
- We shall run it and observe its behaviour, via API hooking.
- We want to know about *all* its behaviour, not just some of it.
- We want to know not only what it does, but also *how* it was composed.
Programs Use System Libraries To Get Work Done

Even HelloWorld uses a system API...
API Hooking Overview

- WIN32 API
- SAMPLE.EXE
- C
- B
- A
- main
- D
- OpenFile
- ExtendFile
- WritePipe
- HOOKS
- LOGGING
- SAMPLE.EXE

Maclean (APL/UW)  APIs for API Hookers  OSDF 2013
API Hooking Obstacles

- ADV32API: 676 functions
- KERNEL32: 963 functions (Common)
- WINSOCK2: 105 functions (Obscure)

A: Maclean (APL/UW)

APIs for API Hookers

OSDF 2013
API Hooking Obstacles

- Huge number of entry points into the Win32 API.
- Hook *coverage* is ratio of all hooked calls to all possible calls.
- Unhooked calls allow the malware to fly under the radar of the hooking system.
- Goal is to maximise hook coverage and hence monitoring power.
Hooking Calls To A Windows Function — Socket

```c
SOCKET socket( int af, int type, int protocol );
```

To monitor this function being called, a hook function might then be

```c
SOCKET socketHOOK( int af, int type, int protocol ) {
    SOCKET result = socketREAL( af, type, protocol );
    logThisCall( "socket", result, af, type, protocol );
    return result;
}
```

An example hook enabling system is Microsoft Detours:

```c
SOCKET (*socketREAL)( int, int, int ) = socket;
DetoursAttach( &socketREAL, socketHOOK );
```
Hooking Calls To A Windows Function — Socket

```c
SOCKET socket( int af, int type, int protocol );
```

To monitor this function being called, a hook function might then be

```c
SOCKET socketHOOK( int af, int type, int protocol ) {
    SOCKET result = socketREAL( af, type, protocol );
    logThisCall( "socket", result, af, type, protocol );
    return result;
}
```

An example hook enabling system is Microsoft Detours:

```c
SOCKET (*socketREAL)( int, int, int ) = socket;
DetoursAttach( &socketREAL, socketHOOK );
```
Hooking Calls To A Windows Function — Socket

SOCKET socket( int af, int type, int protocol );

To monitor this function being called, a hook function might then be

SOCKET socketHOOK( int af, int type, int protocol ) {
    SOCKET result = socketREAL( af, type, protocol );
    logThisCall( "socket", result, af, type, protocol );
    return result;
}

An example hook enabling system is Microsoft Detours:

SOCKET (*socketREAL)( int, int, int ) = socket;
DetoursAttach( &socketREAL, socketHOOK );
Hooking Calls To A Windows Function — Socket

SOCKET socket( int af, int type, int protocol );

To monitor this function being called, a hook function might then be

SOCKET socketHOOK( int af, int type, int protocol ) {
    SOCKET result = socketREAL( af, type, protocol );
    logThisCall( "socket", result, af, type, protocol );
    return result;
}

An example hook enabling system is Microsoft Detours:

SOCKET (*socketREAL)( int, int, int ) = socket;
DetoursAttach( &socketREAL, socketHOOK );
Hooking Calls To A Windows Function — Socket

```c
SOCKET socket( int af, int type, int protocol );
```

To monitor this function being called, a hook function might then be

```c
SOCKET socketHOOK( int af, int type, int protocol ) {
    SOCKET result = socketREAL( af, type, protocol );
    logThisCall( "socket", result, af, type, protocol );
    return result;
}
```

An example hook enabling system is Microsoft Detours:

```c
SOCKET (*socketREAL)( int, int, int ) = socket;
DetoursAttach( &socketREAL, socketHOOK );
```
Hook Function Code Generation Strategies

Each function to be hooked requires its own hook. No re-use possible.

Options for writing a hook for each, every call into the WinAPI:

- Inspect the header files or web docs (MSDN) and transcribe.
  - What the original developer did, for however many calls used.
  - 10 mins to code each hook times 2000+ functions = long time!

- Transcribe entire API to a database and auto-generate from there. Moves the problem rather than solves it.

- Consider the header files to be the database, generate the hooking code from these directly.
Introducing WinC (as in wink, not wince)

WinC is a combination of Java and Windows C:

- A Windows header file parser.
  - Turns C source code into Java objects.
  - Extracts all function declarations and typedefs.
  - Based on Antlr and a contributed C grammar (adapted!)

- A Java API (3 main classes) for hook function code generation.

- A C API (10 functions) for runtime call logging, distribution.
WinC API Hooking Workflow

- Turn the Windows header files into a data structure (parsing).
- Interrogate this data structure to mass produce hook function source code (code generation).
- Instrument the hooks with precise logging of each call (logging).
- Build and deploy the hooks via e.g. DLL injection.
- Collect the logging messages and analyze.
WinC Header File Parser In Action

First, create a one line C file, e.g. `winsock.c`:

```c
#include <Winsock2.h>
```

Next, compile this file on Windows (VizStudio/cygwin/mingw):

```
cl /P winsock.c  // produces winsock2.i, over 1MB!
```

With `#defines` and `#includes` now gone, `winsock2.i` contains just

- Function declarations.
- New types (structs, unions, enums) and typedefs.

Finally, run WinC’s header file parser on `winsock2.i`:

```
winCParser winsock2.i winsock2.tu
```
WinC Header File Parser Result

winCParser winsock2.i winsock2.tu

Located FunctionDeclarations = 2775
Located Typedefs = 2613

- Each C function declaration in the source becomes a Java object.
- Each typedef also becomes a Java object.
- The output object is a *TranslationUnit* (from the C grammar)
- A TranslationUnit is a pair: `List<FunctionDeclaration>, TypedefSystem`
- Save the TU for later use, using e.g. Java serialization.
Hook Function Generation API — TranslationUnit

After the parse phase, we load the saved TU and move on to hook function code auto-generation:

class TranslationUnit {
    static TranslationUnit load( File serializedTU );
    // all function declarations
    List<FunctionDeclaration> funcDecls;
    // all typedefs
    TypedefSystem typedefs;
}

TranslationUnit tu = TranslationUnit.load(
    new File( "winsock2.tu" ) );
for( FunctionDeclaration fd : tu.funcDecls ) {
    autoGenerateHook( fd );
}
Hook Generation API — Functions and Parameters

class FunctionDeclaration {
    String getName();
    void setName(String newName);
    boolean returnsVoid();
    String returnType();
    String declaration();
    String callExpression();
    List<ParameterDeclaration> getParameters();
    String assignableVariable(String name);
}

class ParameterDeclaration {
    String getName();
    boolean isValue();
    boolean isString();
}
Hook Generation API In Action

```java
SOCKET socket( int af, int type, int );

Interrogate the FunctionDeclaration object `fd` created by the parser:

```java
String s = fd.getName();
fd.setName( s + "HOOK" );
print( fd.declaration() );
print( fd.returnType() );
fd.setName( s + "REAL" );
print( fd.callExpression() );
```

```java
SOCKET socketHOOK( int af, int type, int poppy3 ){
    SOCKET result;
    result = socketREAL( af, type, poppy3 );
    return result;
}
```
Hook Generation API In Action

SOCKET socket( int af, int type, int );

Interrogate the FunctionDeclaration object \( \text{fd} \) created by the parser:

```java
String s = fd.getName();
fd.setName( s + "HOOK" );
print( fd.declaration() );
print( fd.returnType() );
fd.setName( s + "REAL" );
print( fd.callExpression() );
```

```java
SOCKET socketHOOK( int af, int type, int poppy3 ){
    SOCKET result;
    result = socketREAL( af, type, poppy3 );
    return result;
}
```
Hook Generation API In Action

SOCKET socket( int af, int type, int );

Interrogate the FunctionDeclaration object \( fd \) created by the parser:

```java
String s = fd.getName();
fd.setName( s + "HOOK" );
print( fd.declaration() );
print( fd.returnType() );
fd.setName( s + "REAL" );
print( fd.callExpression() );
```

```java
SOCKET socketHOOK( int af, int type, int poppy3 ){
    SOCKET result;
    result = socketREAL( af, type, poppy3 );
    return result;
}
```
Hook Generation API In Action

SOCKET socket( int af, int type, int );

Interrogate the FunctionDeclaration object fd created by the parser:

String s = fd.getName();
fd.setName( s + "HOOK" );
print( fd.declaration() );
print( fd.returnType() );
fd.setName( s + "REAL" );
print( fd.callExpression() );

SOCKET socketHOOK( int af, int type, int poppy3 ){
    SOCKET result;
    result = socketREAL( af, type, poppy3 );
    return result;
}
Hook Generation API In Action

SOCKET socket( int af, int type, int );

Interrogate the FunctionDeclaration object \( fd \) created by the parser:

String s = fd.getName();
fd.setName( s + "HOOK" );
print( fd.declaration() );
print( fd.returnType() );
fd.setName( s + "REAL" );
print( fd.callExpression() );

SOCKET socketHOOK( int af, int type, int poppy3 ) {
    SOCKET result;
    result = socketREAL( af, type, poppy3 );
    return result;
}
Hook Generation API In Action

```java
SOCKET socket( int af, int type, int );

Interrogate the FunctionDeclaration object fd created by the parser:

String s = fd.getName();
fd.setName( s + "HOOK" );
print( fd.declaration() );
print( fd.returnType() );
fd.setName( s + "REAL" );
print( fd.callExpression() );

SOCKET socketHOOK( int af, int type, int poppy3 ){
    SOCKET result;
    result = socketREAL( af, type, poppy3 );
    return result;
}
```
Hook Generation API In Action

SOCKET socket( int af, int type, int );

Interrogate the FunctionDeclaration object \( \text{fd} \) created by the parser:

String \( s \) = fd.getName();
fd.setName( \( s + "\text{HOOK}" \) );
print( fd.declaration() );
print( fd.returnType() );
fd.setName( \( s + "\text{REAL}" \) );
print( fd.callExpression() );

SOCKET socketHOOK( int af, int type, int poppy3 ){
    SOCKET result;
    result = socketREAL( af, type, poppy3 );
    return result;
}
Hook Generation API In Action

```java
SOCKET socket( int af, int type, int );

Interrogate the FunctionDeclaration object fd created by the parser:

String s = fd.getName();
fd.setName( s + "HOOK" );
print( fd.declaration() );
print( fd.returnType() );
fd.setName( s + "REAL" );
print( fd.callExpression() );

SOCKET socketHOOK( int af, int type, int poppy3 ){
    SOCKET result;
    result = socketREAL( af, type, poppy3 );
    return result;
}
```
Hook Function Generation Results

winCHookGen winsock2.tu hooks.c

Located Function Declarations = 2775
Printed Source Line Count = 17523

I use Detours for testing, so the code generator produces
  * All the hook functions.
  * All the assignable variables.
  * A hooks table for table-driven hook insertion.

Finally, take `hooks.c` back to Windows to build the DLL.
Instrumenting The Hooked Call

The whole purpose of API hooking is to watch the program in action.

- Want to record the parameters passed in.
- Want to record the call result.
- Also would like to characterize the call site.

```c
SOCKET socketHOOK( int af, int type, int poppy3 ) {
    SOCKET result = socketREAL( af, type, poppy3 );

    // Need to record/transmit parameters, result

    return result;
}
```
print( "int retAddr;" );
print( "\_\_asm mov eax, [ebp+4]; mov retAddr, eax" );
print( "LogSite(" + retAddr + ")" );
print( "LogName(" + fd.getName() + ")" );

SOCKET socketHOOK( int af, int type, int poppy3 ) {
    SOCKET result = socketREAL( af, type, poppy3 );

    int retAddr;
    \_\_asm { mov eax, [ebp+4]; mov retAddr, eax }
    LogSite( retAddr );
    LogName( "socket" );
}

Can further use the return address value to *not* log within-API calls (earlier red arrows).
Logging The Function Call

```c
print( "int retAddr;" );
print( "_asm mov eax,[ebp+4]; mov retAddr,eax" );
print( "LogSite(" + retAddr + ")" );
print( "LogName(" + fd.getName() + ")" );

SOCKET socketHOOK( int af, int type, int poppy3 ) {
    SOCKET result = socketREAL( af, type, poppy3 );

    int retAddr;
    _asm { mov eax, [ebp+4]; mov retAddr, eax }
    LogSite( retAddr );
    LogName( "socket" );
}
```

Can further use the return address value to not log within-API calls (earlier red arrows).
Logging The Function Call

print( "int retAddr;" );
print( "\_asm mov eax,\[ebp+4\]; mov retAddr,eax" );
print( "LogSite(" + retAddr + ")" );
print( "LogName(" + fd.getName() + ")" );

SOCKET socketHOOK( int af, int type, int poppy3 ) {
    SOCKET result = socketREAL( af, type, poppy3 );

    int retAddr;
    \_asm { mov eax, [ebp+4]; mov retAddr, eax } 
    LogSite( retAddr );
    LogName( "socket" );
}

Can further use the return address value to *not* log within-API calls (earlier red arrows).
Logging The Function Parameters

```c
int arity3( DWORD p1, char* p2, struct S* p3);

void logParam( ParameterDeclaration pd ) {
    String s = pd.getName();
    if( pd.isValue() )
        print( "LogValue( sizeof( " + s + "),&" + s + ")" );
    else if( pd.isString() )
        print( "LogString(" + s + ")" );
    else
        print( "LogPointer( sizeof(" + s + "," + s + ")" ) );
}
```

LogValue( sizeof( p1 ), &p1 ); // int, void*
LogString( p2 ); // char*
LogPointer( sizeof( *p3 ), p3 ); // int, void*
Logging The Function Parameters

```c
int arity3( DWORD p1, char* p2, struct S* p3 );

void logParam( ParameterDeclaration pd ) {
    String s = pd.getName();
    if( pd.isValue() )
        print( "LogValue( sizeof( \"+ s + \"),&\" + s + \")\" );
    else if( pd.isString() )
        print( "LogString(\" + s + \")\" );
    else
        print( "LogPointer( sizeof(\" + s + \",\" + s + \")\" );
}

LogValue( sizeof( p1 ), &p1 );   // int, void*
LogString( p2 );                // char*
LogPointer( sizeof( *p3 ), p3 ); // int, void*
```
Logging The Function Parameters

```c
int arity3( DWORD p1, char* p2, struct S* p3 );

void logParam( ParameterDeclaration pd ) {
    String s = pd.getName();
    if( pd.isValue() )
        print( "LogValue( sizeof( " + s + " ),&" + s + " )" );
    else if( pd.isString() )
        print( "LogString(" + s + ")" );
    else
        print( "LogPointer( sizeof(" + s + "," + s + ")" );
}

LogValue( sizeof( p1 ), &p1 );  // int, void*
LogString( p2 );               // char*
LogPointer( sizeof( *p3 ), p3 ); // int, void*
```
Logging The Function Parameters

```c
int arity3( DWORD p1, char* p2, struct S* p3 );

void logParam( ParameterDeclaration pd ) {
    String s = pd.getName();
    if( pd.isValue() )
        print( "LogValue( sizeof( " + s + "),&" + s + ")" );
    else if( pd.isString() )
        print( "LogString(" + s + ")" );
    else
        print( "LogPointer( sizeof(" + s + "," + s + ")" );
}

LogValue( sizeof( p1 ), &p1 ); // int, void*
LogString( p2 ); // char*
LogPointer( sizeof( *p3 ), p3 ); // int, void*
```
Logging The Function Parameters

```c
int arity3( DWORD p1, char* p2, struct S* p3 );

void logParam( ParameterDeclaration pd ) {
    String s = pd.getName();
    if( pd.isValue() )
        print( "LogValue( sizeof( " + s + ") , &" + s + ");
    else if( pd.isString() )
        print( "LogString(" + s + ");
    else
        print( "LogPointer( sizeof(" + s + ", " + s + ");
}

LogValue( sizeof( p1 ), &p1 ); // int, void*
LogString( p2 ); // char*
LogPointer( sizeof( *p3 ), p3 ); // int, void*
```
Resolving Typedef Declarations

```c
void DeleteRegistryEntry( LPSTR key );
```

The parameter appears to be a simple value, and we might log it incorrectly. But following its typedef chain reveals it to be a string:

```c
typedef char CHAR;
typedef CHAR* LPSTR;
```

```c
void DeleteRegistryEntry( char* key );
```

Correct logging of each parameter is thus:

```c
for( ParameterDeclaration pd:fd.getParameters() ) {
    ParameterDeclaration pd2 = tu.typedefs.resolve(pd);
    logParam( pd2 );
}
```
Resolving Typedef Declarations

```c
void DeleteRegistryEntry( LPSTR key );
```

The parameter appears to be a simple value, and we might log it incorrectly. But following its typedef chain reveals it to be a string:

```c
typedef char   CHAR;
typedef CHAR* LPSTR;
```

```c
void DeleteRegistryEntry( char* key );
```

Correct logging of each parameter is thus:

```c
for( ParameterDeclaration pd : fd.getParameters() ) {
    ParameterDeclaration pd2 = tu.typedefs.resolve(pd);
    logParam( pd2 );
}
```
Completed Hook Function

```c
int arity3( DWORD p1, char* p2, struct S* p3 );

int arity3HOOK( DWORD p1, char* p2, struct S* p3 ) {
    int result = arity3REAL( p1, p2, p3 );
    LogValue( sizeof( p1 ), &p1 );
    LogString( p2 );
    LogPointer( sizeof( *p3 ), p3 );
    LogResult( sizeof( result ), &result );
    return result;
}
```
int arity3( DWORD p1, char* p2, struct S* p3 );

void LogPointer( int dataSize, void* ptr2Data ) {
    // dereference ptr2Data, grab dataSize bytes...
}

Should also grab the value of the pointer, it tells us something about the calling program:

- Structure allocated globally.
- Structure allocated on the heap.
- Structure allocated on the stack (avenue for overflow?).

With info from a memory map (Ollydbg), can then fingerprint the coding style, in addition to the runtime behaviour.
Collecting the Logged Information

- Run an agent to harvest hook/log outputs (NamedPipe)
- Forward to central collector (UDP), oversees a whole network?
- Collector archives the log messages. Visualizations too.
- Collector uses same TranslationUnit information to decode the log messages.
- Knowledge about the hooked calls at both sender and receiver mean the log message format is general (and terse), needs no markup.

```
<table>
<thead>
<tr>
<th>LogMessage</th>
<th>Collector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hook</td>
<td>Hook Runtime</td>
</tr>
<tr>
<td>Generation</td>
<td></td>
</tr>
<tr>
<td>TranslationUnit</td>
<td></td>
</tr>
</tbody>
</table>
```
Conclusions, Future Work

- To maximize API hooking effectiveness, need automated hook generation.
- Once have such automation, easy to experiment with different logging strategies.
- Rich, precise logging can fingerprint original coding styles.
- But don’t forget, API hooking is easy to combat, decoy.

Plan to release to github. Looking for testers!