Design Pattern Recovery from Malware Binaries

Cory F. Cohen

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

DM-0002840

Automated Binary Analysis Challenges

Software Assurance

- We need to answer basic questions about functionality
- Does it contain known bad or suspicious code?
- Does this binary program do what we think it does?

Malware Analysis

- Time consuming and complex manual process
- Requires highly specialized reverse engineering skills
- We need to fully automate malware analysis tasks
- Custom tools must be built on a solid foundation

Binary Static Analysis Infrastructure

Components needed for binary analysis framework

- File format parsing
- Disassembler
- Function partitioner
- Instruction semantics
- Emulation framework
- Use-def chains
- SMT solver integration
- Algebraic simplification

We built on the ROSE platform:

- Binary analysis capabilities
- Working closely with LLNL
- BSD Licensed
- C++ Library Implementation
- Highly extensible

We extended ROSE with:

- Calling convention detection
- Stack delta analysis
- Parameter tracking
- Type recovery (in progress)

Objdigger: Object Oriented Analysis

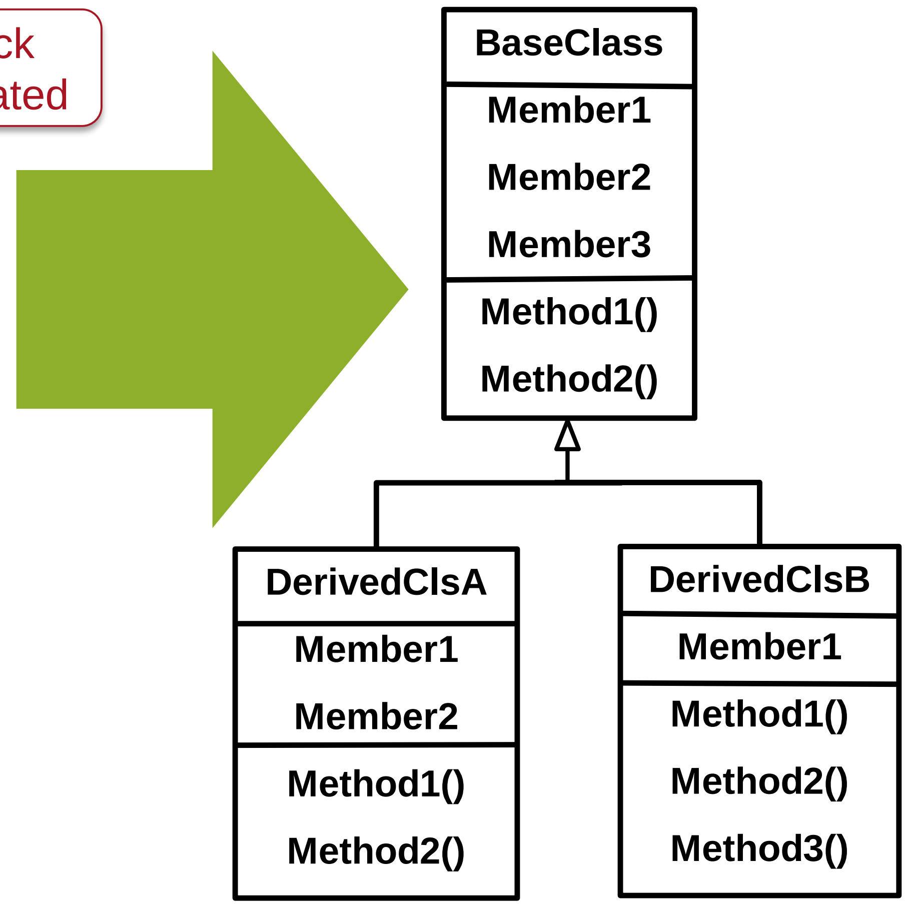
```
00401010 _main          proc near
00401010
00401010 var_B4         = byte ptr -0B4h
00401010 var_C          = dword ptr -0Ch
00401010 var_4          = dword ptr -4
00401010 argc           = dword ptr  4
00401010 argv           = dword ptr  8
00401010 envp           = dword ptr  0Ch
00401010
00401010                push    -1
00401012                push    41497Bh
00401017                mov     eax, large fs:0
0040101D                push    eax
004                     mov     large fs:0, esp
004                     sub     esp, 0A8h
004                     push    esi
0040102C                lea     ecx, [esp+0B8h+var_B4]
00401030                call    sub_403000
00401035                mov     eax, [esp+0B8h+argv]
0040103C                mov     ecx, [esp+0B8h+argc]
00401043                push    eax
00401044                push    ecx
00401045                lea     ecx, [esp+0C0h+var_B4]
00401049                mov     [esp+0C0h+var_4], 0
00401054                call    sub_401470
00401059                lea     ecx, [esp+0B8h+var_B4]
0040105D                mov     esi, eax
0040105F                mov     [esp+0B8h+var_4], -1
0040106A                call    sub_401F20
0040106F                mov     ecx, [esp+0B8h+var_C]
00401076                mov     eax, esi
00401078                pop     esi
00401079                mov     large fs:0, ecx
00401080                add     esp, 0B4h
00401086                retn
00401086 _main          endp
```

Stack Allocated

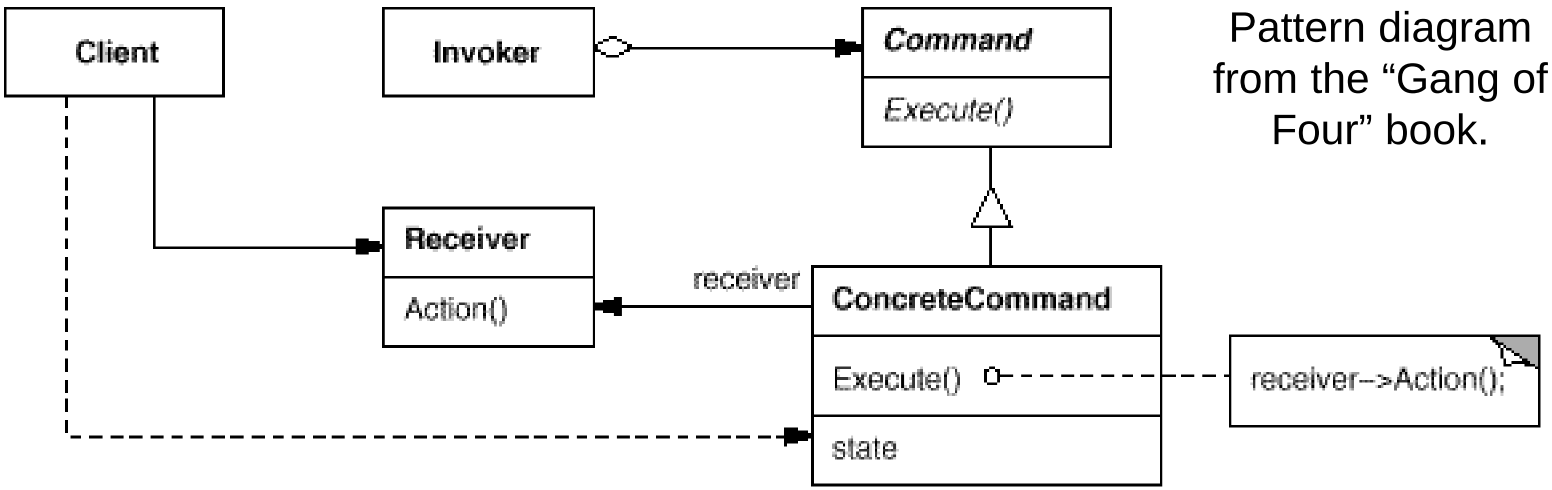
Constructor

Method

Method



Design Pattern Recovery Problem



Pattern diagram from the “Gang of Four” book.

Malware authors face similar software design challenges

- Develop reusable components to ease software evolution
- Combine components in new ways to accomplish goals
- Code reuse is challenged by anti-virus detection efforts

Analysts want to match these patterns in executables

- Recognize higher abstractions in low-level assembly
- Anecdotal evidence supports “malware specific” patterns

A Command Pattern Source Implementation

```
class Receiver {
public:
  void RunCP(PTSTR proc);
  void RunDF(PTSTR filename); };

class Cmd {
public: virtual void Exec() = 0;
protected: Receiver rcvr; };

class Invoker {
public: void runCmd(Cmd& c) {
  c.Exec(); } };

class CPCmd : public Cmd {
private: PTSTR proc;
public:
  CPCmd(Receiver &r, PTSTR p) {
    rcvr = r; proc = p; }
  virtual void Exec() {
    rcvr.RunCP(proc); }
};
```

```
class DFCmd : public Cmd {
private: PTSTR file;
public:
  DFCmd(Receiver &r, PTSTR f) {
    rcvr = r; file = f; }
  virtual void Exec() {
    rcvr.RunDF(file); }
};
```

```
int main() {
  Receiver r;
  CPCmd cp(r, "c:\\calc.exe");
  DFCmd del(r, "mal.txt");
  Invoker i;
  i.runCmd(cp);
  i.runCmd(del);
}
```

A Command Pattern Binary

```
mov   [ebp+this], ecx
mov   ecx, [ebp+this]
call  Cmd_Ctor
mov   eax, [ebp+this]
mov   [eax], offset vftable
mov   eax, [ebp+this]
mov   ecx, [ebp+c]
mov   [eax+8], ecx
mov   eax, [ebp+this]
```

```
mov   [ebp+this], ecx
mov   eax, [ebp+this]
mov   ecx, [eax+8]
push  ecx
mov   ecx, [ebp+this]
add   ecx, 4
call  Receiver_RunCP
```

Example on left is part of CPCmd::CPCmd() on right CPCmd::Exec().

Obviously, many of the source code features are lost or obscured.

But many features are still there as well (as required for execution).

Calling convention identified this pointer, vftable virtual functions, etc.

Features can be extracted using our binary analysis framework.

Design Pattern Features & Detection

Enumerate the features that define the pattern:

1. There exist four unnamed classes (we'll call them C, CC, I, & R).
2. CC inherits from C (begin by temporarily labeling C & CC)
3. The constructor for CC (#2) takes an R as a parameter.
4. There's a method E on CC (#2) that calls a method in R (#3).
5. The method E (#4) is virtual.
6. Class C (#2) contains an instance of R (#3) as a member.
7. Class I that has a method X that takes C or CC (#2) as a parameter.
8. The method X (#7) calls method E (#5).

Test for each feature. Pattern is present if all features are present.

Identified components can be labelled automatically after detection.

Prototype Tool & Experimental Results

We implemented a design pattern matching prototype

- Framework exports facts about program as Prolog facts
- Patterns are very naturally expressed as Prolog rules
- Prolog finds the pattern and reports the matching classes

We conducted an experiment in malware family detection

- Built a gh0st/evilight malware variant from source code
- Detected a variety of classes, methods and functions
- Used class relationships, API sequences, and the call graph
- Core pattern was a socket and a command design pattern
- Primarily leveraged a reciprocal relationship between classes
- Identified command classes both generically and specifically
- Also key constructs like procedural command dispatch loop

Conclusions & Future Research

More work yet to be done on design pattern matching

- Continue to improve accuracy and completeness of features
- Conduct more experiments on pattern variation in malware
- Evaluate expressiveness of patterns given current features
- Evaluate new feature exporters to implement in framework

Successfully detected numerous abstractions in a malware sample

- Allows malware analysts to share knowledge about family
- Reduces effort by assigning semantic labels to abstractions
- Focuses analyst attention on unmatched features in new variants

Future Research in Decompilation

- Focusing on decompilation to source code in FY 2016
- Goal is to allow source analysis tools to be applied to binaries

Questions?

For more information about the Pharos suite of Automated Static Binary Analysis tools, please contact:

Cory Cohen <cfc@cert.org> 1-412-268-7925