Two Tools for Malware Analysis and Reverse Engineering in Ghidra

insights.sei.cmu.edu/blog/two-tools-for-malware-analysis-and-reverse-engineering-in-ghidra/

Jeffrey Gennari

November 1, 2021

Since the public release by the National Security Agency of the software reverse engineering (SRE) tool suite Ghidra, our team of researchers at the SEI's CERT Division have been working to create a new suite of tools to make it easier for analysts to take advantage of Ghirdra's capabilities and interface. This new suite of tools, known as Kaiju, helps malware analysis and reverse engineering take advantage of Ghidra's capabilities and interface, which are relatively easy to use during malware analysis. The tools included with Kaiju give malware analysts many advantages as they are faced with increasingly diverse and complex malware threats.

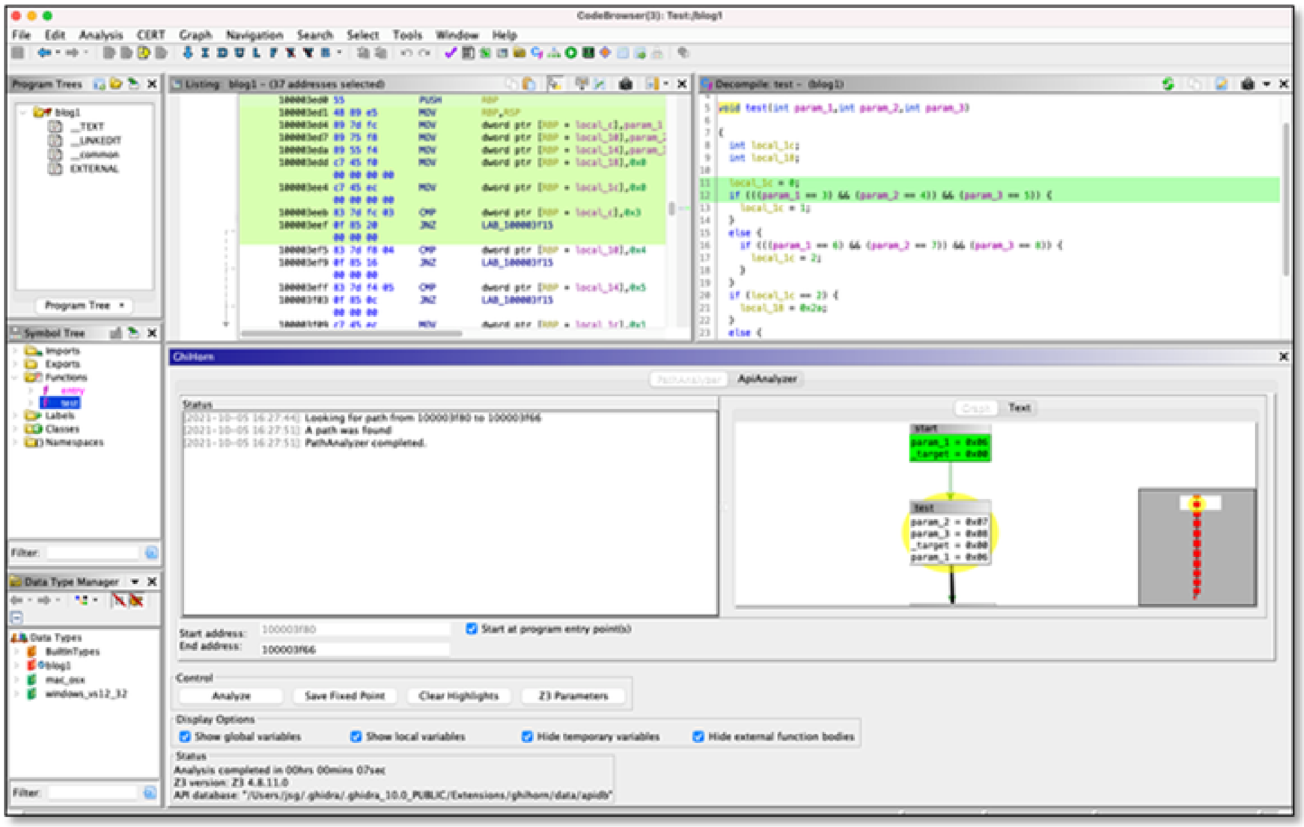
One of the more complex plugins included in Kaiju is a Satisfiability Modulo Theories (SMT)-based path analysis tool named GhiHorn. In this post we present and discuss the use of two Ghihorn tools: API Analyzer and Path Analyzer. Both of these tools enable analysts to find paths in executables based on numerous criteria, such as user-specified start and end addresses or passing through specific program points. Our previous work has shown that executable path finding can support various malware analysis activities. One example that we have previously cited is a malware program that contains a check for the presence of a debugger, a common technique meant to hinder analysis. The analyst may wish to know if there is a viable execution path that circumvents this check and, if there is a path, what inputs and environmental conditions are needed to traverse it. Both PathAnalyzer and ApiAnalyzer can be used to address and solve these types of problems.

GhiHorn Tools: ApiAnalyzer and PathAnalyzer

On top of the GhiHorn platform we have implemented two path analysis tools: *PathAnalyzer* and *ApiAnalyzer*. Both tools use Horn encoding and solving as their primary strategy for path analysis, and both emphasize the usefulness of results.

PathAnalyzer

PathAnalyzer allows an analyst to determine if a path exists between two locations in a program and, if so, what values are required to take that path. PathAnalyzer integrates with Ghidra as a plugin (Figure 1). An analyst must provide start and goal addresses to seed the analysis. If a path is feasible, PathAnalyzer returns an answer in the form of a graph containing information about the recovered path.



```
void test(int param_1,int param_2,int param_3)

{
  int local_1c;
  int local_18;

  local_1c = 0;
  if (((param_1 == 3) && (param_2 == 4)) && (param_3 == 5)) {
    local_1c = 1;
  }
  else {
    if (((param_1 == 6) && (param_2 == 7)) && (param_3 == 8)) {
      local_1c = 2;
    }
  }
  if (local_1c == 2) {
    local_18 = 0x2a;
  }
  else {
```

```
[2021-10-05 16:27:44] Looking for path from 100003f80 to 100003f66
[2021-10-05 16:27:51] A path was found
[2021-10-05 16:27:51] PathAnalyzer completed.
```

Start address: 100003f80
End address: 100003f66
[✓] Start at program entry point(s)

Display Options: [✓] Show global variables [✓] Show local variables [✓] Hide temporary variables [✓] Hide external function bodies

```
Analysis completed in 00hrs 00mins 07sec
Z3 version: Z3 4.8.11.0
API database: "/Users/jsg/.ghidra/.ghidra_10.0_PUBLIC/Extensions/ghihorn/data/apidb"
```

Figure 1: PathAnalyzer user interface

An example of a recovered path is shown in Figure 2. Each vertex in the graph represents a basic block traversed. GhiHorn operates on live variables present in each basic block, with an emphasis on variables that are linked to decompilation data structures. The variable values in the vertices are the values that Z3 selected to reach the goal address. The graph is clickable in the Ghidra user interface, meaning that selecting a node will highlight the relevant decompilation and disassembly instructions for the answer vertex.

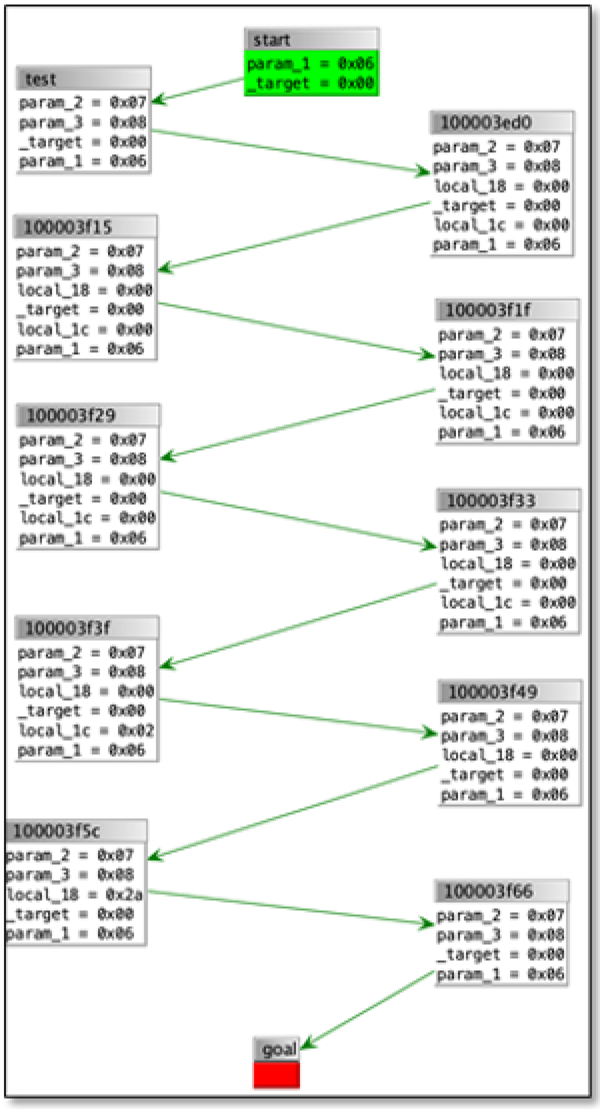
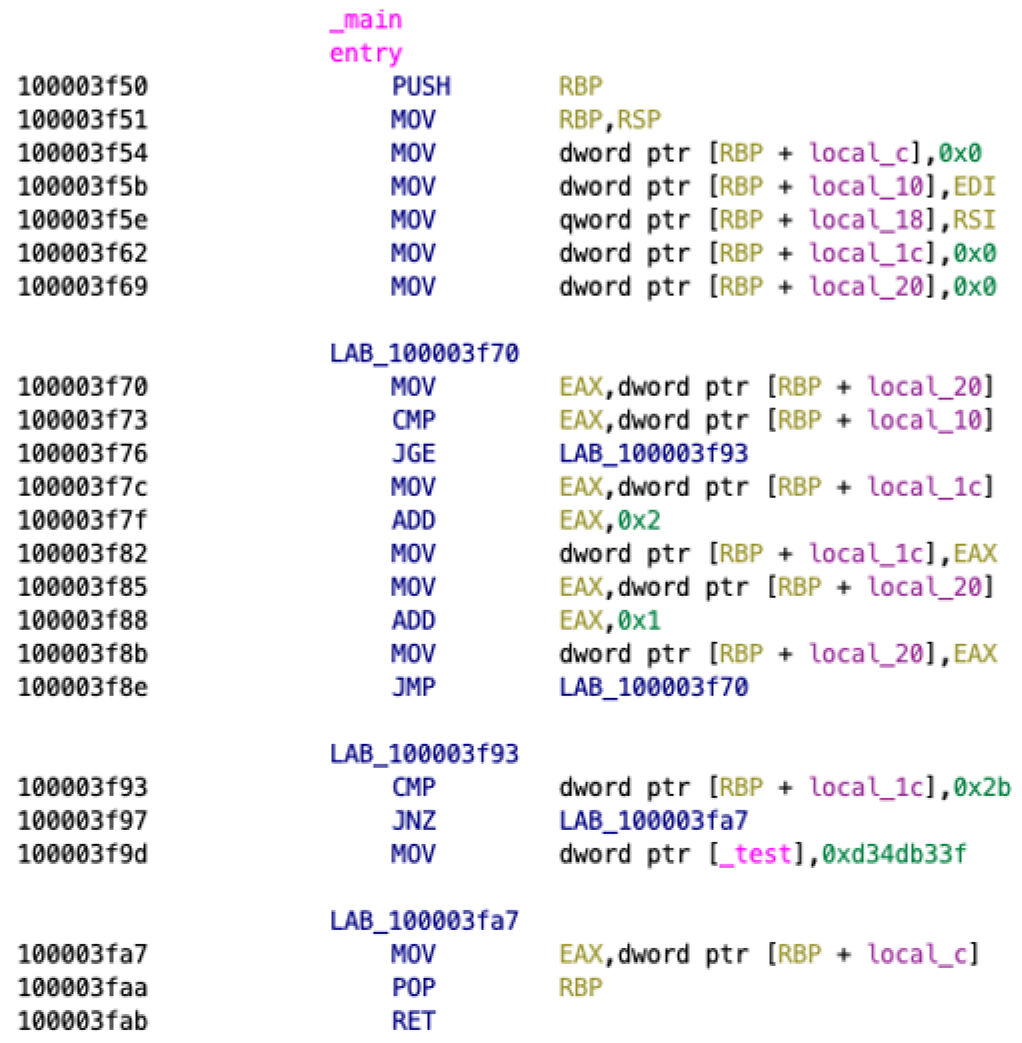


Figure 2: PathAnalyzer answer graph

If no path is found, then a counterexample graph is generated showing how the query failed. In this version of GhiHorn the counterexample is not very meaningful. Currently, the actual results for unsatisfiable queries are presented as they are returned by Z3: as a graph with Boolean values assigned to states found in the encoding. For example, Table 1 shows the disassembly for a function. Attempting to find a path from the beginning of the function at address `0x100003f50` to the address `0x100003f9d` is unsatisfiable because `local_1c` is always an even number, and the condition guarding the instruction at `0x100003f9d` checks for an odd number. The disassembly is shown because the default result for an unsatisfiable answer is to return Boolean values (true or false) for block addresses. Aside from learning that this particular attempt to find a path is infeasible, we are still working to make counterexamples more useful.

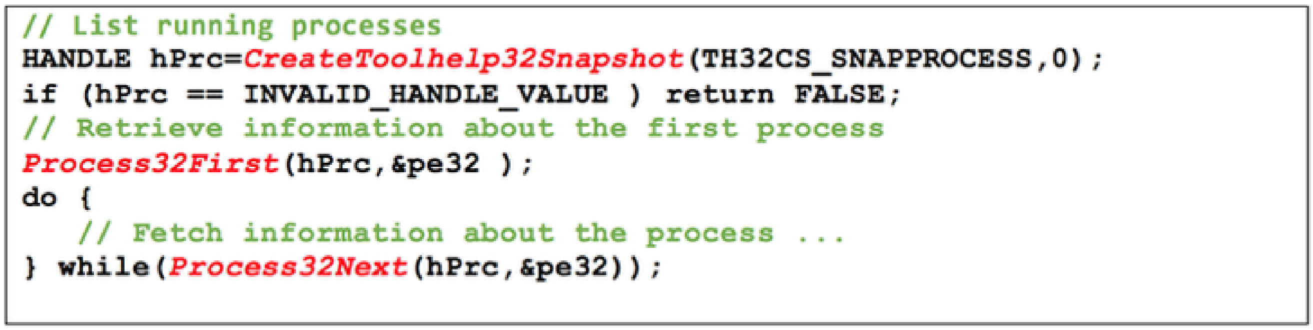


```
                        _main
                        entry
100003f50                   PUSH      RBP
100003f51                   MOV       RBP,RSP
100003f54                   MOV       dword ptr [RBP + local_c],0x0
100003f5b                   MOV       dword ptr [RBP + local_10],EDI
100003f5e                   MOV       qword ptr [RBP + local_18],RSI
100003f62                   MOV       dword ptr [RBP + local_1c],0x0
100003f69                   MOV       dword ptr [RBP + local_20],0x0

                        LAB_100003f70
100003f70                   MOV       EAX,dword ptr [RBP + local_20]
100003f73                   CMP       EAX,dword ptr [RBP + local_10]
100003f76                   JGE       LAB_100003f93
100003f7c                   MOV       EAX,dword ptr [RBP + local_1c]
100003f7f                   ADD       EAX,0x2
100003f82                   MOV       dword ptr [RBP + local_1c],EAX
100003f85                   MOV       EAX,dword ptr [RBP + local_20]
100003f88                   ADD       EAX,0x1
100003f8b                   MOV       dword ptr [RBP + local_20],EAX
100003f8e                   JMP       LAB_100003f70

                        LAB_100003f93
100003f93                   CMP       dword ptr [RBP + local_1c],0x2b
100003f97                   JNZ       LAB_100003fa7
100003f9d                   MOV       dword ptr [_test],0xd34db33f

                        LAB_100003fa7
100003fa7                   MOV       EAX,dword ptr [RBP + local_c]
100003faa                   POP       RBP
100003fab                   RET
```

Table 1: Unsatisfiable example

ApiAnalyzer

The second tool based on GhiHorn, named ApiAnalyzer, uses binary path and data flow analysis to reason about program behaviors. Like its Pharos namesake, the premise of GhiHorn ApiAnalyzer is that interesting program behaviors can be determined by finding API function call sequences that share common data. For example, Figure 3 below shows the common way to list the processes running on a system using the Windows Tool Help Functions. The API calls used are highlighted in red.

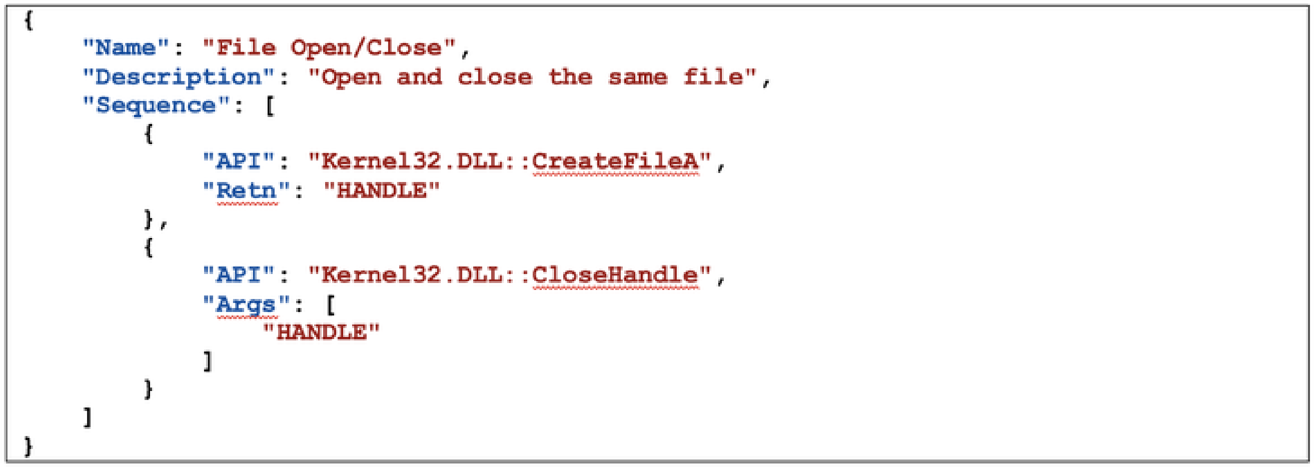


```
// List running processes
HANDLE hPrc=CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS,0);
if (hPrc == INVALID_HANDLE_VALUE ) return FALSE;
// Retrieve information about the first process
Process32First(hPrc,&pe32 );
do {
   // Fetch information about the process ...
} while(Process32Next(hPrc,&pe32));
```

Figure 3: Source code to iterate through running processes using API calls to Windows Tool Help Functions

The GhiHorn version of ApiAnalyzer formulates a program using the same hornified control flow graph as PathAnalyzer. This approach is reasonable given that the API analysis problem can be framed as a path reachability problem with a few additional constraints namely that the path must traverse a specific sequence of API function calls. More specifically the additional constraints are introduced to tally the API function calls covered and to track variable values passed between API functions.

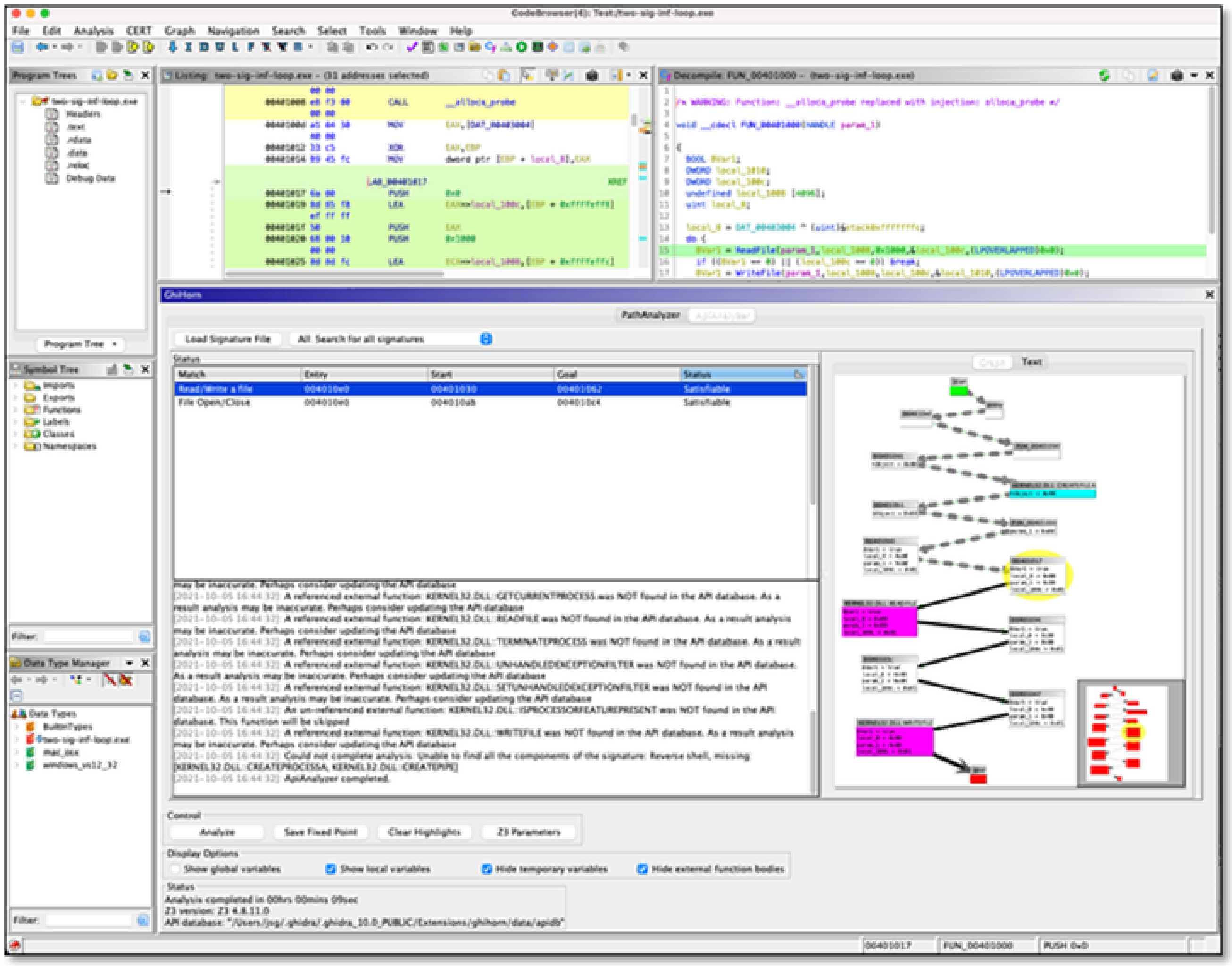
The API call sequences and variables to track are specified as signatures using a JSON-based format similar to that of the Pharos ApiAnalyzer. For example, a simple signature to link the opening and closing of the same file is shown in Table 2. Note that the two functions operate on the same data: a variable labeled `HANDLE. CreateFileA` returns the initial value, and `CloseHandle` must operate on that same value for a match to exist.



```
{
    "Name": "File Open/Close",
    "Description": "Open and close the same file",
    "Sequence": [
        {
            "API": "Kernel32.DLL::CreateFileA",
            "Retn": "HANDLE"
        },
        {
            "API": "Kernel32.DLL::CloseHandle",
            "Args": [
                "HANDLE"
            ]
        }
    ]
}
```

Table 2 : API signature

The ApiAnalyzer plugin is shown in Figure 4. In this image, two simple signatures are being searched for: reading and writing the same file, as well as opening and closing the same file. Both signatures are found in the program, and an answer graph is presented. This graph is shown in Figure 5.



```
/* WARNING: Function: __alloca_probe replaced with injection: alloca_probe */

void __cdecl FUN_00401000(HANDLE param_1)

{
  BOOL BVar1;
  DWORD local_1010;
  DWORD local_100c;
  undefined local_1008 [4096];
  uint local_8;

  local_8 = DAT_00403004 ^ (uint)&stack0xfffffffc;
  do {
    BVar1 = ReadFile(param_1,local_1008,0x1000,&local_100c,(LPOVERLAPPED)0x0);
    if ((BVar1 == 0) || (local_100c == 0)) break;
    BVar1 = WriteFile(param_1,local_1008,local_100c,&local_1010,(LPOVERLAPPED)0x0);
```

Status

Match	Entry	Start	Goal	Status
Read/Write a file	004010e0	00401030	00401062	Satisfiable
File Open/Close	004010e0	004010ab	004010c4	Satisfiable

```
may be inaccurate. Perhaps consider updating the API database
[2021-10-05 16:44:32] A referenced external function: KERNEL32.DLL::GETCURRENTPROCESS was NOT found in the API database. As a
result analysis may be inaccurate. Perhaps consider updating the API database
[2021-10-05 16:44:32] A referenced external function: KERNEL32.DLL::READFILE was NOT found in the API database. As a result analysis
may be inaccurate. Perhaps consider updating the API database
[2021-10-05 16:44:32] A referenced external function: KERNEL32.DLL::TERMINATEPROCESS was NOT found in the API database. As a result
analysis may be inaccurate. Perhaps consider updating the API database
[2021-10-05 16:44:32] A referenced external function: KERNEL32.DLL::UNHANDLEDEXCEPTIONFILTER was NOT found in the API database.
As a result analysis may be inaccurate. Perhaps consider updating the API database
[2021-10-05 16:44:32] A referenced external function: KERNEL32.DLL::SETUNHANDLEDEXCEPTIONFILTER was NOT found in the API
database. As a result analysis may be inaccurate. Perhaps consider updating the API database
[2021-10-05 16:44:32] An un-referenced external function: KERNEL32.DLL::ISPROCESSORFEATUREPRESENT was NOT found in the API
database. This function will be skipped
[2021-10-05 16:44:32] A referenced external function: KERNEL32.DLL::WRITEFILE was NOT found in the API database. As a result analysis
may be inaccurate. Perhaps consider updating the API database
[2021-10-05 16:44:32] Could not complete analysis: Unable to find all the components of the signature: Reverse shell, missing:
[KERNEL32.DLL::CREATEPROCESSA, KERNEL32.DLL::CREATEPIPE]
[2021-10-05 16:44:32] ApiAnalyzer completed.
```

Display Options: Show global variables [✓ Show local variables] [✓ Hide temporary variables] [✓ Hide external function bodies]

Status
Analysis completed in 00hrs 00mins 09sec
Z3 version: Z3 4.8.11.0
API database: "/Users/jsg/.ghidra/.ghidra_10.0_PUBLIC/Extensions/ghihorn/data/apidb"

Figure 4: Ghidra ApiAnalyzer user interface

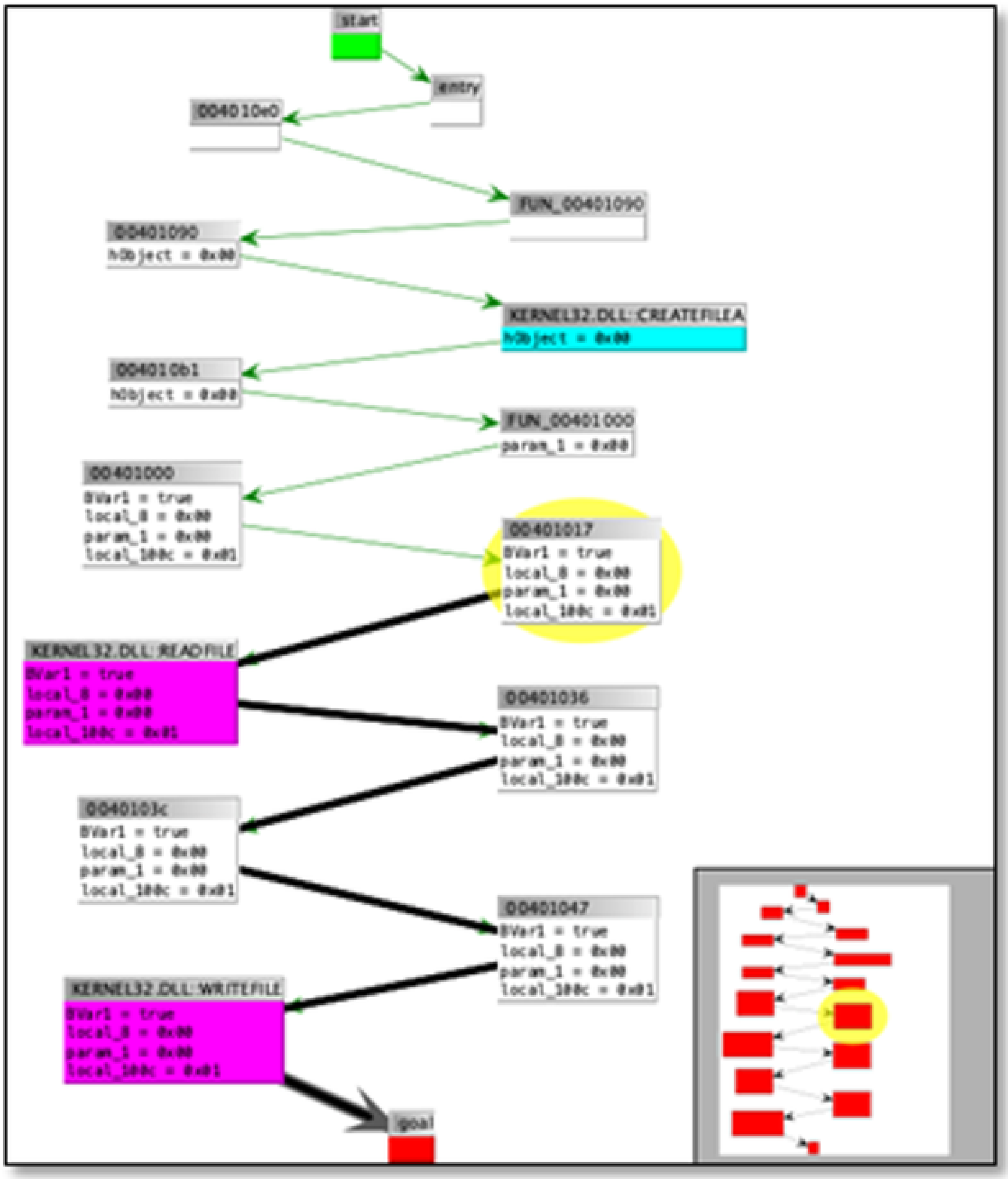


Figure 5: ApiAnalyzer answer graph

ApiAnalyzer's answer graph is basically the same as the graph generated by PathAnalyzer. Imported API functions for which there is an implementation are highlighted in cyan. API functions for which there is no implementation are colored magenta. As before, the graph includes variable values that are used to construct the path, and the graph can be used for navigation in the Ghidra disassembly and decompiler windows.

The Future of GhiHorn

GhiHorn provides the capability to reason about program paths in Ghidra. The Horn-based tools provided by GhiHorn and the tight integration with Ghidra's user interface makes formal program analysis tools accessible to malware analysts and reverse engineers. Future versions of the plugin will continue to provide new and better ways to consume results and to model more complex code structures. We also plan on generating better counterexamples to make it more evident why a path could not be recovered. Finally, we're working on a better memory model that better represents a real program's address space.

GhiHorn is publicly available as part of the CERT Kaiju Framework. The source code and build instructions for GhiHorn (and all Kaiju tools) are available on GitHub. We welcome suggestions for improvements or for new utilities that would be useful for building new tools to support malware analysis and reverse engineering.

Additional Resources

Read the SEI Blog post *Introducing CERT Kaiju: Malware Analysis Tools for Ghidra*.

Read the SEI Blog post *GhiHorn: Path Analysis in Ghidra Using SMT Solvers*.

View the SEI Podcast *Reverse Engineering Object-Oriented Code with Ghidra and New Pharos Tools*.

WRITTEN BY

MORE BY THE AUTHOR

GhiHorn: Path Analysis in Ghidra Using SMT Solvers

October 18, 2021 • By Jeffrey Gennari

Introducing CERT Kaiju: Malware Analysis Tools for Ghidra

September 13, 2021 • By Garret Wassermann , Jeffrey Gennari

Using OOAnalyzer to Reverse Engineer Object Oriented Code with Ghidra

July 15, 2019 • By Jeffrey Gennari

Path Finding in Malicious Binaries: First in a Series

December 10, 2018 • By Jeffrey Gennari

Pharos Binary Static Analysis Tools Released on GitHub

August 28, 2017 • By Jeffrey Gennari

MORE IN REVERSE ENGINEERING FOR MALWARE ANALYSIS

GhiHorn: Path Analysis in Ghidra Using SMT Solvers

October 18, 2021 • By Jeffrey Gennari

Introducing CERT Kaiju: Malware Analysis Tools for Ghidra

September 13, 2021 • By Garret Wassermann , Jeffrey Gennari

3 Ransomware Defense Strategies

November 9, 2020 • By Marisa Midler

Using OOAnalyzer to Reverse Engineer Object Oriented Code with Ghidra

July 15, 2019 • By Jeffrey Gennari

Business Email Compromise: Operation Wire Wire and New Attack Vectors

April 8, 2019 • By Anne Connell