Using Control-Flow Techniques in a Security Context
A Survey on Common Prototypes and their Common Weakness

Mark M. Seeger

Department of Secure Services
Center for Advanced Security Research Darmstadt (CASED)
Mornewegstraße 32, 64293 Darmstadt, Germany
mark.seeger@cased.de

and

Department of Computer Science
Gjøvik University College
N-2818 Gjøvik, Norway

***Abstract*—Practical approaches using control-flow techniques in order to detect changes in the control-flow of a program have been subject of many scientific works.**
This work focuses on three common tools making use of control- and data-flow analysis in order to detect alternations and reveals their common weakness in terms of the ability to react directly to a dynamic change in control-flow. With a general focus on static analysis of binaries or source code, detection of dynamic changes in the executive flow cannot be detected. In order to emphasize this shortcoming of static analysis, we present an approach for dynamically changing a program's control-flow and validate it by depicting a proof of concept.

***Keywords*-Host Intrusion Detection, Control-Flow Analysis, Control-Flow Attack, Late Binding**

I. Introduction

Analyzing the flow of control and data within a program is a well-known technique intrinsic to the field of compiler construction. While program optimization is one of its main fields of application, the field of security with regard to information technology (IT) has chronicled the appearance of control- and data-flow analysis years ago. The appearance of subversion techniques concealing their malicious intents by altering the program flow contributed its part to this development. Nowadays, buffer overflow attacks are the widest spread representative of exploiting the deviation of control-flow. Here, software vulnerabilities are used in order to inject and execute defective code. Thus, detecting such attacks is possible and practical approaches able of countering them have been presented by various authors.
We took a closer look at three common prototypes which make use of flow analyzing techniques in order to detect a change in the executive flow of a program. The result of the investigation is subject of this work and reveals that all of them follow a static concept. That is, they are unable to react to dynamic changes in control-flow.
To this extend, we developed a C++ program making use of a Linux system call mostly utilized to implement breakpoint debugging in order to take advantage of a characteristic of the late binding technique, which is commonly used for loading dynamic link libraries (DLL). In a testing environment, this proof of concept was able to change the control-flow of a target program while staying undetected by major antivirus suits.
The remainder of this work is structured as follows: Section II presents a short history on control- and data-flow techniques in general before Section III goes into detail by outlining the major features of control-flow (III-A) and data-flow (III-B) analysis. The brief presentation of the three selected common prototypes is subject of Section IV. Section V depicts our prototype by describing the approach as such, as well as showing some pseudo code. The paper is closed with a conclusion in Section VI and an outlook into the future work in Section VII.

II. Related Work

Control-flow analysis is not new at all. Under the name of "boolean matrices" it was already mentioned in papers published in the second third of the 20^{th} century with the earliest one being from Lefschetz, published in 1930 [1]. The idea behind this special kind of matrices evolved from the need to have the data of flow diagrams available in such a way that it could be handled by machines. Likewise to today's ambitions, this data was used for program analysis with regard to internal consistency and the identification of subroutines [2].
Extensive research has been done in this field with regard to the applicability of control-flow analysis in a security related context. In general, the control-flow of a given program is studied in order to tell apart normal from abnormal behavior; to say this with the words of Forrest et

al.: “The problem of protecting computer systems can be viewed generally as the problem of learning to distinguish *self* from *other*.” [3]. With respect to this, Wespi et al. ([4]) proposed a technique for creating patterns that can be used to model normal behavior of a given process. These models represent the *self* and can be used for intrusion detection under the assumption that the *other* can be identified by a behavior different from what was used to train the model.

Enforcing control-flow for security reasons is – amongst others – the subject of [5] and [6]. While the former present a binary rewriting approach in order to augment existing programs, the latter present an architectural mechanism in which the processor tracks the information flows in question and therefore, is capable of detecting dangerous uses of spurious values.

Abadi et al. claim that the enforcement of control-flow integrity (CFI) “...cannot be subverted or circumvented even though it applies to the inner workings.” [7]. In their paper, they show the practical prove that CFI enforcement is capable of ensuring that runtime execution of a given program proceeds along a given control-flow graph[1] (CFG). While for intrusion detection purposes both techniques are used, control-flow and data-flow analysis, [9] shows how the gap between them can be bridged. They present practical results on how data-flow analysis can be used in order to leverage the results from control-flow analysis.

In [10] the authors demonstrate that static analysis for intrusion detection is no longer a sufficient technique for identifying malware. They use binary code obfuscation in order to circumvent detection by semantics-based malware scanners.

III. Flow Analysis

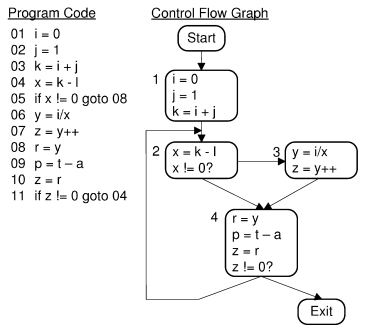
Flow analysis comes in all kind of flavors – with and without relevance to information security. In this section we focus on the two different techniques of control-flow and data-flow analysis. While both techniques were initially used for optimization reasons (and still are), they are nowadays also used in the field of intrusion and malware detection.

As we will see, the two presented analyzing techniques are not mutually exclusive as one can be used as a preprocessor for the other.

A. Control-Flow Analysis

From a very abstract point of view, a program is nothing else than a series of instructions that can be addressed. The decision as to which instruction is to be executed next depends on the result of what is currently happening. By analyzing the source code of a program or its binary, all instructions can be identified and a CFG containing all possible flows of control can be retrieved. A CFG contains

[1]A software called “Vulcan” [8] was used to gain the control-flow graphs.



Program Code

```
01  i = 0
02  j = 1
03  k = i + j
04  x = k - l
05  if x != 0 goto 08
06  y = i/x
07  z = y++
08  r = y
09  p = t – a
10  z = r
11  if z != 0 goto 04
```

Figure 1. Example of a Control-Flow Graph.

nodes and edges. Each node represents what is called a basic block: a maximum set of instructions without any possibility to merge or fork. There is an edge between two nodes if the last instruction of a basic block leads to the start of another basic block. A simple version of a CFG is shown in Figure 1. As in a CFG there is usually more than one possible path leading from the start of the program to its final exit, an automated way of analyzing this structure is needed. This is the subject of control-flow analysis.

B. Data-Flow Analysis

As the name already suggests, data-flow analysis is rather concerned with the data that flows within a program than its control. With respect to information security, the meaningfulness of CFGs has been proven in theory and practice and its analysis is known to be effective. Data-flow analysis on the other hand is a discipline whose history started in 1977 when Hecht published his book on *Flow Analysis of Computer Programs* [11].

Data-flow relates to the arguments of system calls and to the actual data (i.e. variables) that a specific program deals with. With respect to information security, monitoring the system call arguments in contrast to analyzing CFGs of a given program, comes with the advantage that malicious behavior can be detected even in cases where the actual control-flow stays unmodified.

Generally data-flow analysis is concerned with the question of the lifetime of variables. As the states of all variables may never be known and since the input of one basic block is the output of its predecessor, data-flow analyses is done by approximating the values of variables coming into a specific basic block. The calculation is then done by feeding these values into iterative algorithms such as the round robin algorithm in order to find so called fix points. That is, values for the incoming variables that satisfy the function $f : X \to X$.

Using data-flow analysis for anomaly detection is the subject of [12]. In their work, the authors present an algorithm that can be layered on top of existing programs that rely on control-flow techniques for being invulnerable against attacks such as non-control-flow hijacking attacks where, for

instance, configuration data or user input is used in order to exploit vulnerabilities of certain applications.
Figure 2 shows a sample data-flow graph (DFG). For a better comparability between CFG and DFG, the DFG is based on the same synthetic program code as the CFG shown in Figure 1.

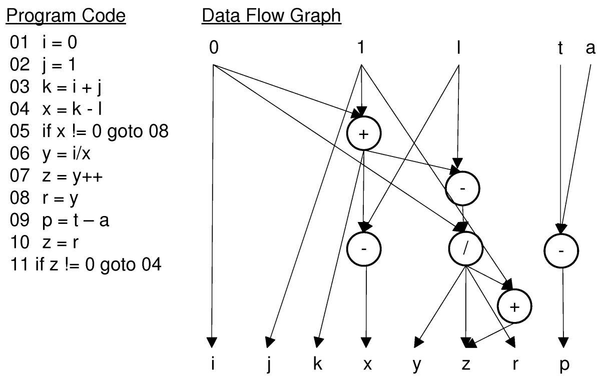


Figure 2. Example of a Data-Flow Graph.

IV. Common Prototypes Using Flow Analyzing Techniques in a Security Context

In this section we present three common software prototypes which use flow analyzing techniques for security purposes.

A. Binary Executable Structurizer

In [13] a "lightweight assembler structural representation tool" called BEST (Binary Executable Structurizer) is introduced. It incorporates the use of control- and data-flow analyzing techniques to analyze the executive flow of a program. As the name already suggests, BEST does not rely on the source code of the programs in question as it works with the program binaries. Besides the recovery of the control- and data-flow, the tool also represents the assembler of a binary in a "vulnerability-mining-oriented intermediate language" called PANDA. In contrast to plain assembler, a PANDA representation of assembler code is easier to assess and thus, facilitates further security analysis. Since PANDA's basic control structures as well as mathematical and logical operations are taken from the C programming language, a program represented in PANDA looks much like a C program with some embedded assembler syntax.
In order to retrieve the flow graphs from a binary, BEST relies on third party programs, namely IDA Pro ([14]) and Objdump ([15]). Based on the output of these tools (either one or the other is used), the BEST algorithms start working. Besides the PANDA code, BEST also outputs a control-flow graph and a control tree graph (shows the nesting relationship of control structures). With the PANA code available, security professionals can then assess the binary in question in order to unveil security flaws.

B. Control-Flow Security Analysis Approach

In [16] a control-flow security model (CFSM) which allows program execution to dynamically follow the statically declared security properties specified as a control-flow constraint specification (CFCS) is introduced. This approach counters, for instance, buffer overflow attacks, which have the intention of diverting program execution by jumping to previously injected malicious code. Thus, illegally altering the execution flow of programs in order to control the software behavior. The proposed model includes the formal definition for program semantics and security properties for control-flow. With the CFSM applied to source code or binaries, execution paths violating the predefined CFCSs can be identified and prevented. This is done by ensuring two security properties which regard to data memory that is not executable and code memory that is not writable. In the latter case, observations due to loading dynamic link libraries are excluded.
The control-flow graph needed to define the CFSM is retrieved applying a two step approach starting with a conservative CGF based on the analysis of function calls and the branch jumps of basic blocks, followed by a static analysis to refine an improve the former outcome. The overall result is a graphical control-flow security model annotated with the corresponding control-flow constraints.

C. Static Analyzer for Executables

In addition to an architecture that is resilient to common obfuscation transformations and capable of detecting malicious patterns in x86 executable binaries, [17] also introduces the proof of their concept by implementing SAFE, a static analyzer for executables. The heart of SAFE – namely the detection component – takes two arguments. One is an annotated control-flow graph of the executable and the other is a generalization of malicious code (i.e. a representation of obfuscated strains of a virus). The CFG itself is the result of the annotation module, which uses preloaded pattern definitions to annotate the plain CFG gained from a combinational use of the third party tools IDA Pro ([14]) and CodeSurfer ([18]). While Norton Antivirs, McAfee VirusScan and Command Antivirus were not able to detect any of the four obfuscated[2] viruses during the practical experiments, SAFE was able to identify 100% of them and thus, has a false rate of zero.

D. Review of the Presented Prototypes

In the preceding subsections, we gave a survey of three common prototypes which are using flow analyzing techniques in a security related context. While BEST and the CFSM approach are from 2010 and 2009, respectively, SAFE was presented in 2003. Table I gives an overview over five main characteristics of the tools presented in Section IV. All of them use control-flow techniques while only BEST

[2]Obfuscation technique: code transposition

Table I
CHARACTERISTICS OF THE PRESENTED FLOW ANALYZING TOOLS.

Tool Name	Uses CFG	Uses DFG	Operates Statically	Reads Binary	3^{rd} Party Tools
BEST	x	x	x	x	x
CDSM	x	o	x	x	o
SAFE	x	o	x	x	x

combines control- and data-flow. Also, all tools perform a static analysis on binary executables. Only the CFSM approach waives of 3^{rd} party tools while both BEST and SAFE depend on the quality of the outcome of tools such as IDA Pro ([14]), CoderSurfer ([18]) or Objdump ([15]).
BEST is not a fully automated tool. Its result is a "vulnerability-mining-oriented intermediate language" [13] of the binary in question, which serves the purpose of easing the process of program code understanding. Thus, reacting to ongoing security breaches is not possible.
The CFSM approach is able to detect any exploit that alters the execution paths of the binary in question. While the approach presented is a reasonable method in order to detect buffer overflow attacks, it comes with one major restriction: Alterations made directly or indirectly due to loading dynamic link libraries cannot be detected. SAFE, which is able to detect obfuscated viruses with a much higher rate of reliability than Norton Antivirs, McAfee VirusScan and Command Antivirus relies on both the quality of the output of 3^{rd} party tools and the completeness of the pattern definitions. All three approaches have one drawback in common: They are not designed to instantly react to changes in the executional flow caused by dynamically loaded code.

V. EXAMPLE OF DYNAMICALLY ALTERING CONTROL-FLOW

In this section, we will present our approach which is capable of dynamically altering the execution flow of arbitrary executables[3] using late binding techniques for loading dynamic link libraries. The according proof of concept is implemented as three C++ files. Listing 1 depicts the target program which is the subject of our attack[4]. In line 6, we create a function pointer pointing to a function named `isValidUser()`, implemented in a dynamic link library. This function is called in line 8. In line 2 of our DLL shown in Listing 2, the `userID` of the current user is checked. Only if it is lower than 1,000 a function named `doAdminStuff()` is executed. The original control-flow is depicted as solid-lined arrows in Figure 3.

[3]While our proof of concept presented here regards to the Linux operating system, we also have an equivalent version running on Windows.
[4]In the field of offline computer games, software that implements such features in order to alter game specific values (e.g. health, energy, ammunition, etc.) is called a trainer.

Listing 1. Pseudo Code of the Target Program.
```
 1 main()
 2 {
 3     int   ret               = 0;
 4     char *dllName           = "myDll";
 5     void *handle            = dlopen(dllName);
 6     int (*isValidUser)()    = dlsym(handle, "isValidUser");
 7
 8     ret = isValidUser();
 9 }
```

Listing 2. Pseudo Code of the DLL.
```
 1 extern int isValidUser()
 2 {
 3     if (userId >= 1000) {
 4         return 1;
 5     }
 6     doAdminStuff();
 7     return 0;
 8 }
 9
10 void doAdminStuff()
11 {
12     ...
13 }
```

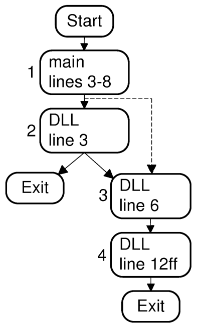


Figure 3. CFG of the Target Program (Lst. 1) and its DLL (Lst. 2).

Listing 3. Pseudo Code the Program Executing the Exploit.
```
 1 main(int argc, char *argc[])
 2 {
 3     int pid          = argc[0];
 4     long oldPointer = argc[1];
 5     long newPointer = argc[2];
 6
 7     ptrace(PTRACE\_ATTACH, pid, NULL, NULL)
 8     ptrace(PTRACE\_POKEDATA, pid, oldPointer, newPointer);
 9     ptrace(PTRACE\_DETACH, pid, NULL, NULL);
10 }
```

The exploit code is shown in Listing 3. It is started with three parameters, namely the process id, the address of the function to be attacked (`oldPointer`) and the address of the new function (`newPointer`). `oldPointer` is the address the function pointer `(*isValidUser)()` which points to the address of the variable on the heap and `newPointer` is the address of the new function (`doAdminStuff()`). In order to achieve our goal of altering the control-flow dynamically, we make use of the `ptrace()` system call, that "...provides a means by which a parent process may observe and control the execution of another process, and examine and change its core image and registers." [19]. This method is mainly used for implementing breakpoint debugging and system call tracing. In line 7 we start tracing the target process. As a result of this, we are from now on the parent of the target. While the output of the `ps()` command reflects this change in ownership, issuing the `getppid()` command would reveal the original parent. Before we are undoing the reparenting effect in line 9, we perform the actual attack in line 8: Here, we overwrite the address of the old function with the address of the function we want to execute instead. With this being done, we dynamically changed the control-flow. The function call in line 8 of Listing 1 does not point to line 1 of Listing 2 anymore, thus, the check of the user id in line 3 is circumvented. Instead function `doAdminStuff()` is executed directly. The altered control-flow is depicted as a dashed-lined arrow in Figure 3, completely bypassing basic block 2. This attack works and stays undetected because it takes advantage of a specific characteristic of the late binding technique. When late binding is used, the affected variables

are stored on the heap. They stay writable since they receive their actual value during runtime. Our approach exploits this fact and uses a system call intended for debugging purposes in order to overwrite the value of a specific variable stored on the heap.
Being able to detect such attacks is a matter of distinguishing "*self* from *other*" [3].

VI. Conclusion

We reviewed three common practical tools which make use of control-flow graphs in a security related context. Specific to all of them is the fact that they operate statically and thus, are unable to react to dynamically undertaken changes of control-flow. To this extend, we have presented a proof of concept taking advantage of a Linux system call, mainly used for debugging purposes, and a characteristic of the late binding technique, where the corresponding variables are placed on the heap. We showed pseudo source code of our exploit and pointed out that altering the control-flow of a program which make use of the late binding technique (e.g. loading of DLLs) is possible. Countering such an attack is not an easy task since it does not use any vulnerability. Therefore, counter measurements are fronted with the problem of telling apart legal alterations of the value of variables stored in the heap and illegal ones – the subject of anomaly detection.

VII. Future Work

Our future work will be twofold. We will be concerned with developing a proof of concept which will perform observation tasks on variables on the heap. In compliance with a predefined security policy, any changes on values ultimately resulting in a change of executive flow will be brought to the attention of the user who will be given the ability to allow or block the action. In addition to that, we will concentrate on a novel concept allowing for automatically made decisions regarding the degree to which an alteration of control-flow regards to *self* oder *other* (i.e. is legal of illegal).

Acknowledgment

The author would like to thank Julian Knauer for his excellent practical support.

References

[1] S. Lefschetz, "Topology," *American Mathematical Society*, vol. XII, 1930.

[2] R. T. Prosser, "Applications of Boolean Matrices to the Analysis of Flow Diagrams," *International Workshop on Managing Requirements Knowledge*, 1959.

[3] S. Forrest, A. S. Perelson, L. Allen, and R. Cherukuri, "Self-Nonself Discrimination in a Computer," in *IEEE SP'94*, 1994.

[4] A. Wespi, H. Debar, M. Dacier, and M. Nassehi, "Fixed- vs. variable-length patterns for detecting suspicious process behavior," *J. Comput. Secur.*, 2000.

[5] M. Prasad and T. Chiueh, "A binary rewriting defense against stack based overflow attacks," in *In Proceedings of the USENIX Annual Technical Conference*, 2003.

[6] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," *ASPLOS-XI*, 2004.

[7] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM TISSEC*, 2009.

[8] A. Srivastava, A. Edwards, and H. Vo, "Vulcan – Binary transformation in a distributed environment," Microsoft Research, Tech. Rep., 2001.

[9] P. Li, H. Park, D. Gao, and J. Fu, "Bridging the Gap between Data-Flow and Control-Flow Analysis for Anomaly Detection," in *ACSAC 2008*, 2008.

[10] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *ACSAC 2007*, 2007.

[11] M. S. Hecht, *Flow Analysis of Computer Programs*. Elsevier Science Inc., 1977.

[12] S. Bhatkar, A. Chaturvedi, and R. Sekar, "Dataflow Anomaly Detection," in *In Proc. IEEE Symposium on Security and Privacy*, 2006.

[13] W. Wang, "Best: An assembler structural representation tool based on flow analysis," in *MASS 2010*, 2010.

[14] Hex-Rays. (2010, Nov.) IDA Pro. [Online]. Available: http://www.hex-rays.com/idapro

[15] GNU Binutils. (2010, Nov.) ObjDump. [Online]. Available: www.gnu.org/software/binutils

[16] W. Chunlei, Z. Gang, and D. Yiqi, "An efficient control flow security analysis approach for binary executables," in *IEEE ICCSIT 2009*, 2009.

[17] M. Christodorescu and S. Jha, "Static analysis of executables to detect malicious patterns," in *12th SSYM'03*, 2003.

[18] Grammatech. (2010, Nov.) CodeSurfer. [Online]. Available: www.grammatech.com/products/codesurfer

[19] Linux Manpage. (2010, Nov.) ptrace(). online. [Online]. Available: http://linux.die.net/man/2/ptrace