# Data is Flowing in the Wind: A Review of Data-Flow Integrity Methods to Overcome Non-Control-Data Attacks

Irene Díez-Franco, Igor Santos

DeustoTech, University of Deusto
Bilbao, Spain
{irene.diez,isantos}@deusto.es

**Abstract.** Security researchers have been focusing on developing mitigation and protection mechanisms against *code-injection* and *code-reuse* attacks. Modern defences focus on protecting the legitimate control-flow of a program, nevertheless they cannot withstand a more subtle type of attack, *non-control-data* attacks, since they follow the legitimate control flow, and thus leave no trace. *Data-Flow Integrity* (DFI) is a defence mechanism which aims to protect programs against non-control-data attacks. DFI uses static analysis to compute the data-flow graph of a program, and then, enforce at runtime that the data-flow of the program follows the legitimate path; otherwise the execution is aborted.
In this paper, we review the state of the techniques to generate non-control-data attacks and present the state of DFI methods.

**Keywords:** data-flow integrity, non-control-data attacks, operating system security

## 1 Introduction

Programs are formed by control data (e.g. return addresses, function pointers) and non-control data (e.g. variables, constants). Even though non-control data is more abundant, both attackers and defenders have focused their efforts into exploiting or protecting control data. Different attacks have been repeatedly applied to the control flow, such as complex *code-injection* and *code-reuse* attacks. Therefore, memory integrity methods have been proposed in order to secure operating systems and userland programs (e.g. safe dialects of C/C++, secure virtual architectures, and control-flow integrity methods).

Even though non-control-data attacks are not new [8] and their importance has not decreased, modern operating systems and their programs remain vulnerable against these type of attacks. Given the lack of usable methods against these attacks, in this paper we present a review of the most relevant techniques to overcome non-control-data attacks, as well as a overview of the landscape of non-control-data attacks.

## 2 Non-Control-Data Attacks

The most common memory corruption vulnerabilities, namely *code-injection* and *code-reuse* attacks, have been tackled by the community, creating defences for commodity operating systems and compilers. On the one hand, stack canaries [9] and write-xor-execute (W⊕E)/DEP [3] defence schemes try to prevent *code-injection* attacks resulting from stack, heap or buffer overflows, use-after-free and format string vulnerabilities, whereas Control-Flow Integrity (CFI) [1] and program shepherding [14], along with the statistical defences provided by ASLR [20] and Kernel ASLR [11] concentrate on *code-reuse* attacks arising from classic return-oriented programming (ROP) [15, 17], or its newer variants [7, 4, 5].

Due to the attention given to code-injection and code-reuse attacks, Chen et al. [8] raised awareness of the *pure data* or *non-control-data* attacks, since all the previous approaches [9, 3, 1, 14, 20, 11] focus just on *control-data*, and thereby cannot endure more subtle non-control-data attacks.

A non-control-data attack differs from a control-data attack because it does not affect the control-flow of a program. Control-data attacks are based on rewriting control data (e.g. return addresses), leaving a trace in the form of an unintended control-flow transfer. This transfer can be detected and prevented at runtime [1].

On the contrary, non-control-data attacks follow legitimate control-flow transfers since they are based on modifying the program's logic or decision-making data. Consequently, they remain invisible to defence techniques which only focus on control data.

### 2.1 Security-Critical Non-Control Data

Chen et al.'s work identifies the following types of security-critical data that may be subjected to non-control-data attacks:

- **Configuration data**. Many applications, such as web servers, need configuration files in order to define access control policies and file path directives to specify the location of trusted executables. If an attacker was capable of overwriting such configuration data, it would be possible to launch unintended applications (e.g. root shells), and moreover bypass the access controls of the web server.
- **User input data**. A well known practice in software engineering is to distrust user input data, and only after that data has been validated it can be used. If an attacker could change the input data after the validation process, she could execute the program with a malicious input.
- **User identity data**. UIDs and GIDs are stored in memory while authentication routines are executed. If such IDs were tampered with, an attacker could impersonate a user with administration privileges.
- **Decision-making data**. Boolean values (e.g. authenticated or not) are usually used to make decisions in an application, an attacker could change those

```
struct passwd {
  uid_t pw_uid;
  ...
} *pw;
...
int uid = getuid();
pw->pw_uid = uid;
// format string vulnerability
...
void passive(void) {
  ...
  seteuid(0); // set root uid
  ...
  seteuid(pw->pw_uid); // set non-root uid
}
```

**Fig. 1.** Vulnerable code from the `wu-ftpd` web server.

decision making values to redirect the flow of a program through unintended branches.

Moreover, Hu et al. [12] enhanced the previously presented types of security-critical data with the following items:

- **Passwords and private keys**. The disclosure of passwords and private keys could give an attacker full access to a system.
- **Randomised values**. Tags for CFI enforcement, random canary words and randomised addresses are used in many security related mechanisms (e.g. CFI, ASLR, SSP). If an attacker knows the random canaries placed in the stack she can use stack-smashing attacks without alerting the Stack Smashing Protector (SSP).
- **System call parameters**. Tampering with the parameters of security critical system calls (e.g. `execve`, `setuid`) can lead to privilege escalation or unintended program execution.

The following two attacks describe how a vulnerability can be exploited due to a memory error to expose security-critical non-control data using a non-control-data attack.

Figure 1 shows a vulnerable piece of code with a format string vulnerability where the value of `pw->pw_uid` can be rewritten. This vulnerability can be exploited using a non-control-data attack that targets *user identity data*. Concretely, an attacker could overwrite the value with root's UID, to, later on avoid dropping root privileges to normal user privileges in the `passive` function.

The OpenSSL *Heartbleed* vulnerability [21] allowed a remote attacker to expose sensitive data, such as *private keys*, using a non-control-data attack. On the OpenSSL 1.0.1 and 1.0.2$\beta$ heartbeat request/response protocol, an attacker could request a heartbeat using legitimate payload but with a payload length

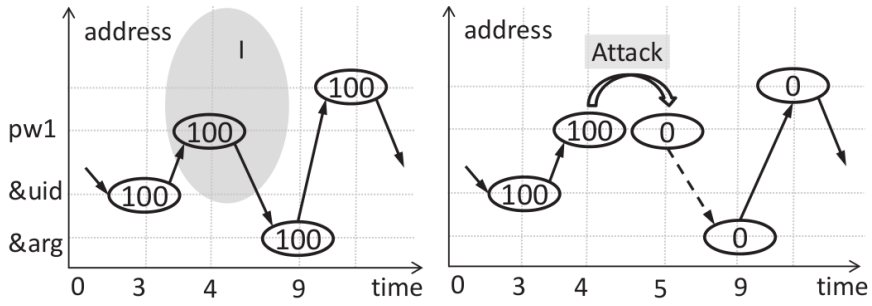


**Fig. 2.** Original 2D-DFG of the vulnerable code of `wu-ftpd` web server (left) and resulting 2D-DFG after a non-control-data attack (right). `&arg` is the stack address of `setuid`'s argument.

field larger (up to 65,535 bytes) than the real payload. Then, the heartbeat protocol crafted a response copying the original payload in a buffer allocated of the size indicated by the payload length field. Since the payload length field was not correctly verified against the length of the real payload, a memory leakage was possible.

### 2.2 Data-Flow Stitching

Hu et al. [12] demonstrated that non-control-data attacks can be automatically constructed using a technique named *data-flow stitching*. This technique is capable of redirecting the data-flow of a program through unintended paths in order to tamper with data or leak sensitive data.

They introduced the concept of *two-dimensional data-flow graph* (2D-DFG) to represent the data dependencies created in a program executed with a concrete input. A 2D-DFG is a directed graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$ where $\mathcal{V}$ is the set of vertices and $\mathcal{E}$ the set of edges. A vertex $v$ ($v \in \mathcal{V}$) is a variable instance with a value, and it is represented as a point $(a, t)$ in the two dimensions of the 2D-DFG, addresses and time; thereby, $a$ refers to the address or register name of the variable, and $t$ to the execution time when the variable instance is created. A *vertex* $v = (a, t)$ is created when an instruction writes to memory value $a$ at time $t$; a *data edge* $(v', v)$ is created when the instruction takes $v'$ as the source and $v$ as the destination operands, and finally, an *address edge* $(v', v)$ is created when an instruction uses $v'$ as the address of the operand $v$.

In order to generate a non-control-data attack, data-flow stitching requires a program with a memory error. The set of memory locations this memory error can write to is called the *influence* $I$. The new data-flow that wants to be created consists of two vertices, namely source vertex ($v_s$), and target vertex ($v_t$); resulting in a data-flow path from $v_s$ to $v_t$. This new data-flow path would have a new 2D-DFG $\mathcal{G}' = \{\mathcal{V}', \mathcal{E}'\}$, where $\mathcal{V}'$ and $\mathcal{E}'$ are generated by the memory error exploit. The goal of data-flow stitching is to discover a data-flow edge set

$\overline{\mathcal{E}}$, where $\overline{\mathcal{E}} = \mathcal{E}' - \mathcal{E}$, to add in the resulting 2D-DFG of the memory error ($\mathcal{G}'$), allowing new data-flow paths from $v_s$ to $v_t$.

Following the examples given by Chen et al. and Hu et al., Figure 1 shows a vulnerable piece of code from the `wu-ftpd` web server, and Figure 2 (left) shows the original 2D-DFG of such program. The format string vulnerability on `wu-ftpd` server can overwrite `pw->pw_uid`, since such vertex is under the influence of the memory error; thereby, a privilege escalation attack is possible using a non-control-data attack generated by data-flow stitching that inserts an edge in the DFG, overwriting `pw->pw_uid` with root's UID, as shown in Figure 2 (right).

This example requires only the addition of a single edge in the new 2D-DFG, nevertheless data-flow stitching can also generate advanced attacks that need stitching more edges.

### 2.3 Data-Oriented Programming

*Data-oriented programming* (DOP) is a technique proposed by Hu et al. [13] to perform computations on a program's memory respecting its legitimate control-flow. DOP's computations are based on non-control-data attacks resulting from memory errors and have been proven to be Turing-complete. DOP is comparable to the computations made using gadgets on code-reuse attacks by ROP [17], JOP [7, 4], and sigreturn-oriented programming [5]; however, unlike all the previous approaches, DOP is based on *data-oriented gadgets* that have a small number of differences compared to classic gadgets.

DOP requires the use of (i) *data-oriented gadgets* and (ii) a *gadget dispatcher*. On the one hand, data-oriented gadgets form the virtual instructions (i.e. arithmetic, logical, assignment, load, store, jump and conditional jump) required to simulate a Turing machine. Data-oriented gadgets can simulate these operations using the x86 instruction set the same way ROP and JOP do. In contrast, data-oriented gadgets need to use just memory and not memory or registers to generate its operations. In addition, data-oriented gadgets must follow the legitimate control flow. On the contrary, one of the benefits of data-oriented gadgets is that they can be scattered and consequently, there is no need for them to be executed one after the other.

On the other hand, the gadget dispatcher allows the chaining of the data-oriented gadgets and simulating control operations. This dispatcher allows an attacker to choose the sequence of data-oriented gadgets (e.g. creating loops), resulting in *interactive* and *non-interactive* DOP attacks. Interactive attacks use loops and at every loop iteration, a selector controlled by the memory error selects the sequence of data-oriented gadgets that must be executed. Non-interactive attacks require a single payload where all the data-oriented gadgets must be chained.

## 3 Data-Flow Integrity

Castro et al. introduced *data-flow integrity* (DFI) [6], a defensive technique that aims to protect programs against non-control-data attacks. DFI targets x86 architecture ensuring that a given program's data stays within the permitted paths. Firstly, DFI generates the data-flow graph (DFG) of the program by static analysis, secondly, it instruments the program introducing data-flow integrity checks, and finally, it enforces at runtime that the data-flow of the program is allowed by the DFG, otherwise the execution is aborted.

DFI relies on *reaching definitions analysis* [2] for the static DFG generation. Reaching definitions analysis is a static analysis technique used by modern compilers to deploy global code optimisation (e.g. dead code elimination), based on data-flow analysis. Data-flow analysis tries to extract information about the flow of data from program execution paths [2], and reaching definitions analysis concretely deals with the definition (i.e. assignment) and use (i.e. read) of variables. Using reaching definitions analysis, DFI can compute a DFG that contains a set of definitions, assigns an identifier to each definition, and maps those identifiers to instructions. In this way, the DFG shows the instructions that assigned a value to each used variable.

DFI uses two different static analyses to generate reaching definitions, (i) a *flow-sensitive intra-procedural* analysis and (ii) a *context-insensitive inter-procedural* analysis. The former is used to compute the reaching definitions of variables that have no definition outside the function in which they are declared, whereas the later computes the reaching definitions of variables with definitions outside the function in which they are declared. This separation is done to increase the performance of the analysis, since flow-insensitive algorithms have less computing overhead.

Once the static DFG has been generated, DFI instruments at runtime the program to check before every variable use that its definition is within the statically generated reaching definitions identifiers. If not, the data-flow integrity property does not hold, and the program must be terminated. Castro et al.'s approach makes use of a *runtime definitions table* (RDT) to keep track of the last definition of each identifier. In order to check if the data-flow integrity holds, the last value of the RDT for a given identifier must be checked against the static DFG.

In order to be effective, DFI itself must remain safe against sabotages, requiring (i) the integrity of the RDT, (ii) the integrity of the code, and (iii) the integrity of DFI's instrumentation. RDT integrity is achieved ensuring that the definitions are within the memory boundaries defined for the RDT, code integrity is accomplished using modern operating systems' W$\oplus$E check on pagination. The integrity of the instrumentation performed by DFI can be ensured relying on DFI (e.g. instrumenting uses and definitions of control-data made by the compiler) or using additional defences, such as CFI [1] and program shepherding [14].

### 3.1 Kernel Data-Flow Integrity

The operating system is the first line of defence against attacks based on memory corruption on userland applications. However the OS itself is not safe against non-control-data attacks. If an attacker were capable to successfully gain control of the OS, all the defences deployed in userland applications would become futile.

Song et al. [18] utilise DFI in order to enforce kernel security invariants related to access control mechanisms against memory-corruption attacks. They proposed a system named KENALI in order to protect two security invariants, (i) *complete mediation*, attackers have to be prevented from bypassing access control checks, and (ii) *tamper proof*, the integrity of the data and code of the reference monitors must be maintained.

As to enforce these invariants KENALI uses two techniques: *InferDist* and *ProtectDists*. InferDist is used to distinguish the *distinguished regions*, which are the regions that have essential data for enforcing the security invariants. ProtectDists enforces DFI over these regions and due to invariant (i) complete mediation, CFI must also be enforced.

Furthermore, InferDist uses the kernel CFI mechanisms proposed by Criswell et al. [10] to protect control-data. Regarding non-control data, KENALI enforces that if a security check fails, it will return the `-EACCESS` error code (permission denied). InferDist retrieves these error codes and, through dependency analysis on the conditional variables of the security checks, is able to discover which are the distinguished regions.

Finally, to enforce the DFI over the inference result regions, KENALI distinguishes tree types of data-flow (i) within non-distinguishing regions, (ii) between two different types of regions and (iii) within distinguishing regions. KENALI protects the distinguishing regions using a two-layer scheme. The first layer is a lightweight data-flow isolation mechanism to protect the second type of data-flow (between two different types of regions), and a more heavy DFI enforcement mechanism when the two regions are distinguishing.

## 4 Related work

Apart from DFI, *Dynamic Information Flow Tracking* (DIFT) [19] and *Dynamic Taint Analysis* are two techniques that can be applied to prevent non-control-data attacks. DIFT is a hardware mechanism to track malicious information flows. These information flows are controlled by the operating system which denies the usage of suspicious paths for the flow of information through suspicious paths.

DTA [16] is a technique to monitor taint sources of a program while it is executing. DTA is commonly used in malware analysis and vulnerability discovery, and thereby it can be applied to non-control-data attack detection.

However, these techniques are not the focus of this review paper since they have not been applied yet to prevent non-control-data attacks.

## 5 Discussion and conclusions

Non-control-data attacks are gaining more importance due to the efforts that have been directed into preventing and mitigating control-data attacks. Despite all the tools and methods deployed to protect commodity operating systems and userland programs against code-injection and code-reuse attacks, there is still a lack of security mechanisms to protect the same operating systems and userland programs against non-control-data attacks. We believe the main challenge of DFI is to overcome the trade off between computing overhead and completeness, because the use of more accurate and thus more slow static analysis techniques in the DFG generation can prevent users from using these tools.

## References


1. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity: Principles, Implementations and Applications. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (2005)
2. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2006)
3. Andersen, S., Abella, V.: Data Execution Prevention. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies (2004)
4. Bletsch, T., Jiang, X., Freeh, V.W., Liang, Z.: Jump-oriented programming: a new class of code-reuse attack. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (2011)
5. Bosman, E., Bos, H.: Framing signals-a return to portable shellcode. In: Proceedings of the IEEE Symposium on Security and Privacy (Oakland) (2014)
6. Castro, M., Costa, M., Harris, T.: Securing software by enforcing data-flow integrity. In: Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI) (2006)
7. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.R., Shacham, H., Winandy, M.: Return-oriented programming without returns. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (2010)
8. Chen, S., Xu, J., Sezer, E.C., Gauriar, P., Iyer, R.K.: Non-Control-Data Attacks Are Realistic Threats. In: Proceedings of the USENIX Security Symposium (2005)
9. Cowan, C., Pu, C., Maier, D., Hinton, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In: Proceedings of the USENIX Security Symposium (1998)
10. Criswell, J., Dautenhahn, N., Adve, V.: KCoFI: Complete control-flow integrity for commodity operating system kernels. In: Proceedings of the IEEE Symposium on Security and Privacy (Oakland) (2014)
11. Giuffrida, C., Kuijsten, A., Tanenbaum, A.S.: Enhanced operating system security through efficient and fine-grained address space randomization. In: Proceedings of the USENIX Security Symposium (2012)
12. Hu, H., Chua, Z.L., Adrian, S., Saxena, P., Liang, Z.: Automatic generation of Data-Oriented Exploits. In: Proceedings of the USENIX Security Symposium (2015)

13. Hu, H., Shinde, S., Adrian, S., Chua, Z.L., Saxena, P., Liang, Z.: Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks. In: Proceedings of the IEEE Symposium on Security and Privacy (Oakland) (2016)
14. Kiriansky, V., Bruening, D., Amarasinghe, S.P., et al.: Secure execution via program shepherding. In: Proceedings of the USENIX Security Symposium (2002)
15. Nergal: The advanced return-into-lib(c) exploits: Pax case study. Phrack Magazine 58 (2001)
16. Schwartz, E.J., Avgerinos, T., Brumley, D.: All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask). In: Proceedings of the IEEE Symposium on Security and Privacy (Oakland) (2010)
17. Shacham, H.: The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (2007)
18. Song, C., Lee, B., Lu, K., Harris, W., Kim, T., Lee, W.: Enforcing Kernel Security Invariants with Data Flow Integrity. In: Annual Network and Distributed System Security Symposium (NDSS) (2016)
19. Suh, G.E., Lee, J.W., Zhang, D., Devadas, S.: Secure Program Execution via Dynamic Information Flow Tracking. In: Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (2004)
20. Team, P.: Address space layout randomization (ASLR). `http://pax.grsecurity.net/docs/aslr.txt` (2003)
21. US-CERT: OpenSSL 'Heartbleed' vulnerability (CVE-2014-0160). `https://www.us-cert.gov/ncas/alerts/TA14-098A` (2014)