Identifying Arbitrary Memory Access Vulnerabilities in Privilege-Separated Software

Hong Hu, Zheng Leong Chua, Zhenkai Liang, and Prateek Saxena

Department of Computer Science, National University of Singapore
{huhong, chuazl, liangzk, prateeks}@comp.nus.edu.sg

Abstract. Privilege separation is a widely used technique to secure complex software systems. With privilege separation, software components are divided into several partitions and these partitions can only communicate through limited interfaces. However, the interfaces still provide a channel for one partition to influence code in other partitions. As a result, certain memory access patterns can be leveraged by attackers to perform arbitrary memory access. We refer to this type of memory access errors by the acronym *DUI (Dereference Under the Influence)*. In this paper, we present a systematic method to detect vulnerabilities leading to DUI through binary analysis, and to estimate the capability attackers can obtain through DUI exploits. The evaluation shows that our approach can accurately identify vulnerable code that leads to arbitrary memory access in real-world software components and programs, when they are transformed to privilege-separated designs.

1 Introduction

Privilege separation is widely used to secure complex software systems. With this method, software components are divided into several partitions. Each partition only has a reduced set of privileges and inter-partition communication is only possible via clearly defined interfaces. To protect legacy programs using privilege separation, developers need to transform the monolithic legacy programs. For example, the OpenSSH server was originally implemented as a monolithic program, where a single vulnerability will expose all critical resources to attackers. To mitigate the threat, part of OpenSSH code without access to high-privileged resources (e.g., password) was separated from other code and isolated as a slave process [39]. In addition, Qmail [4], Postfix [7] and Google Chrome [3, 46] are also designed (or re-designed) with privilege separation.

To facilitate retrofitting legacy code into privilege-separation designs, many solutions have been proposed to partition software and assign each partition a different set of privileges, such as Wedge [6], Privtrans [11], and Privman [27]. The deployed techniques include sandboxing [44–46] and process-based isolation [6, 11, 27]. When the monolithic code is divided into several partitions, some program behaviors inside the original code (e.g., function calls or direct memory access) need to be transformed into inter-partition communications (e.g., via socket and shared memory). As a result, program logic ensuring the correctness of program semantics, such as valid ranges of variables, may be separated into different partitions and fail to enforce the correctness. Therefore, additional checking code is often needed, especially in the high-privileged

partitions, to make sure that data from other partitions is valid. However, if the transformation process fails to include checking code in a high-privileged partition, or the added checking code is inadequate, attackers can use specially-crafted inputs to compromise the high-privileged partitions and carry out malicious actions with escalated privileges.

Memory errors such as buffer overflow can be exploited in such cross-partition attacks. There are more subtle memory errors with which attackers can perform arbitrary memory access inside the high-privileged partition, if the victim partition has certain memory access patterns. For example, if a partition uses an input from an untrusted interface as the array index, writing to the array inside the partition is an arbitrary memory access under the influence of the input provider. If attackers provide the input, they can utilize this memory access behavior to modify critical data or retrieve secrets of the partition in a targeted manner.

In this paper, we refer to this type of memory access errors by the acronym *DUI (Dereference Under the Influence)*. It stems from the memory access pattern in the vulnerable partition: The address used in memory read or memory write is influenced by malicious data from other partitions. Through DUI exploits, attackers can corrupt discrete memory locations, instead of a continuous memory block, significantly improving the stability of the attacks.

Challenges. DUI exploits can be prevented through sufficient checks on interface inputs. Unfortunately, it is non-trivial to ensure that adequate checking code has been added at correct locations. The checking code in legacy programs is often scattered across many program locations in the monolithic code base, which is split during the privilege-separation transformation, it is necessary to guarantee that each of these program locations are checked correctly. However, to achieve this goal, manual modification usually takes a long time to fully understand the requirement of checking operations, while automatic separation methods often miss important checks. Therefore, we need a systematic method to detect such DUI vulnerabilities resulting from privilege-separation transformations.

Our Approach. To address these challenges, we develop an approach, DUI Detector, to automatically detect code suspicious to DUI exploits in the binary of trusted partitions. Specifically, through binary program analysis, we identify the suspicious instructions that use data from other partitions to dereference memories. Then we use symbolic analysis to identify code with the DUI vulnerability and assess the attackers' capabilities in exploiting them. DUI Detector helps to identify concrete code instances that are easily influenced by attackers among a large code base.

We applied our approach on several real-world software systems retrofitted with different types of isolation schemes. DUI Detector successfully detected DUI vulnerabilities inside them. We present four case studies where attackers can perform DUI attacks. Furthermore, our approach reports the attackers' capability to the developers, providing a comprehensive understanding of the vulnerability.

In summary, this paper makes the following contributions:

- We study the problem of arbitrary memory access (DUI) in privilege-separation transformations, and identify several general memory access patterns leading to DUI vulnerabilities in binary instruction level.

– We design a novel mechanism to automatically detect DUI vulnerabilities, and to estimate attackers' capability in controlling user memory spaces. It helps developers add sufficient checking.
– We prototype an automated tool and evaluate it on several real-world software. Our tool automatically detects and comprehensively analyzes DUIs in these software programs when they are gone through privilege-separation transformation.

2 Problem Overview

In this section, we motivate the problem by a concrete example. Then we provide the problem definition of DUI detection and discuss two DUI types: The write DUI and the read DUI. At the end we discuss the memory access patterns used to detect DUI vulnerabilities.

```
1  struct subobj { ... } * p_sub;
2  struct object { ...
3     struct subobj * sub;
4  } * p_obj;
5
6  int main()  {
7     p_obj = create_object();
8     p_sub = create_subobj();
9     p_obj->sub = p_sub;
10 }
11
12 // create an object instance and return its pointer
13 struct object * create_object()
14 { ... }
15
16 // create a subobj instance and return its pointer
17 struct subobj * create_subobj()
18 { ... }
```

Listing 1.1. Example code to illustrate DUI problem.

2.1 Motivating Example

We use the example in List 1.1 to illustrate the memory access problem during the program transformation. In this example, the structure `object` has one pointer of structure `subobj`. Functions `create_object` and `create_subobj` return pointers of new structure instances. The statement on line 9 in function `main` assigns the pointer `p_sub` of a `subobj` instance to the `subobj` field of a `object` instance. Originally function `create_object` and `create_subobj` are in the same partition with the function

`main`, and there are checking code inside them to make sure that the return values are correct. During the transformation, these two functions are separated into a low-privileged partition since they are not trusted any more. In this case, the return values could be malicious. To protect the high-privileged `main` function, we can use memory isolation to prevent the direct memory access from low-privileged code to `main`'s memory, in which case function `create_object` and `create_subobj` just manage `main`'s memory. However, this is inadequate to protect the high-privileged `main`: The statement on line 9 contains a memory access vulnerability. When the low-privileged partition is malicious, it allows writing a malicious `p_obj` to a memory location `p_sub` inside the protected one. The statement on line 9 is an instance of DUI vulnerability.

2.2 Problem Definition

We give the definition of the problem solved in this paper.

DUI Detection: Given a partition of a privilege-separated program, we detect whether the partition's memory access behaviors can be influenced by data from its interfaces. In particular, the memory addresses or the data are derived from the interface inputs, giving attackers the ability to read or write to a large range of memory inside the partition.

Attackers use the DUI vulnerability as a memory access service to mount attacks. They specify the address and the data through specially-crafted inputs. DUI vulnerability then finishes the memory operation on behalf of attackers. In real-world programs, the logics used to derive the address from the interface inputs could be complicated, thus subtle and hard to spot. However, the final result is that the attacker can exercise certain levels of influence over the address of the memory operation. It is worthwhile to note that only controlling the memory address is inadequate to corrupt the memory or to steal the sensitive information. Corresponding data flows are necessary to provide the malicious data or send the confidential data out. Based on the direction of the memory access, there are two types of DUI, the write DUI and the read DUI.

```
1  v1 = API_recv();
2  v2 = API_recv();
3  array[v1] = v2;
```

Listing 1.2. An example of write DUI

Write DUI. We call a memory write operation the *write DUI* if both the memory address and the value to be written in the operation are derived from the interface inputs. Take the code in Listing 1.2 as an example. The $API\_recv()$ is an interface through which the code can receive data from other partitions. The memory write operation on line 3 has the address $array + v1$ and the data $v2$ derived from the interface inputs, which allows the input provider to write the selected data to any address in its memory space. We can relax the requirement of the data to be written to a value predictable by attackers. An example is that if $v2$ in Listing 1.2 is a constant 0, attackers can use $v2$ to reset important flags, or terminate a C-style string. With the write DUI, attackers can corrupt the memory of the vulnerable program. Not only can they mount control flow

hijacking attacks by corrupting code pointers or return addresses, they can also change critical data in memory to mount non-control-data attacks [16, 26].

```
1  v1   = API_recv();
2  data = *(base + v1);
3  API_send(data);
```

Listing 1.3. An example of read DUI

Read DUI. We call a memory read operation the *read DUI* if the memory address in the operation is derived from the interface inputs and the retrieved data are eventually passed to the output interface of the partition. Consider the example in Listing 1.3. $API\_recv()$ and $API\_send()$ are APIs used by the code to receive data from other partitions and send data to other partitions, respectively. This code snippet retrieves data from a local buffer and sends it out. Since the data retrieving address $base + v1$ is under the control of attackers via the interface input $v1$, attackers can steal sensitive information from the partition. For read DUI, it is insufficient to control the memory read address. The data being read has to reach an output interface for it to complete. In real-world programs, the web client may have secret keys or high-privileged files on the server client. Attackers can use read DUI vulnerability to steal the secret key or file. Another exploit is to leak the randomized address of the loaded modules, leading to bypassing address randomization protections [5, 37].

2.3 Memory Access Patterns to Detect DUIs

Although attackers can use various ways to control the memory access, one DUI vulnerability is inevitably represented as attacker-controllable memory address and data in memory access instructions, i.e., the address is derived from the input, and the data also comes from the input (for the write DUI) or is sent out (for the read DUI). This observation inspires us to use instruction-level memory access patterns to detect DUI vulnerabilities. We summarize the memory access patterns used in DUI exploits below.

- *Write DUI Pattern 1*. The memory write address and the data are derived from the interface inputs. In this case, attackers control both the value and the address in memory write operation.
- *Write DUI Pattern 2*. The memory write address is derived from the interface inputs. The data value is predictable to attackers. Attackers can exploit this code to set the predictable value to any memory address.
- *Read DUI Pattern*. The memory read address is calculated from the interface inputs. The retrieved data are then passed to output interfaces (e.g., via network, file operation or standard printing).

3 Design

3.1 Overview

Figure 1 shows the design of our DUI detection tool, DUI Detector. It takes two inputs: The program binary containing the partition to be checked and a normal input to the

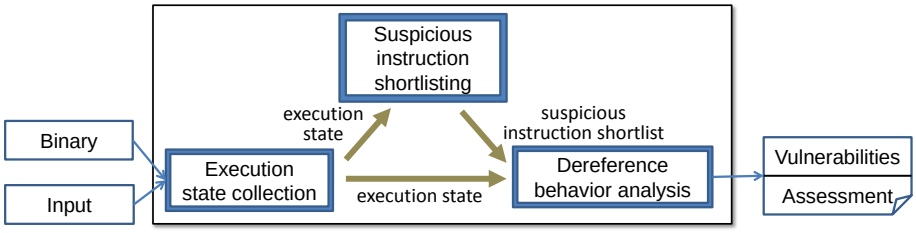

Fig. 1. Design of the DUI Detector. There are two inputs to the system. One is the program binary, containing the partition to be checked. Another one is a normal input to the program. The output is the list of DUI vulnerabilities and the assessment of attackers' capabilities.

program. It detects DUI vulnerabilities during the binary execution for the given input and estimates the capability of attackers obtained by DUI exploits. There are three phases in the process: Execution state collection, suspicious instruction shortlisting and dereference behavior analysis.

Execution State Collection. First we run the program binary in an emulated environment with the given input and record all the execution states of the checked partition, including instructions, operands, processor states and memory states. We also log system-level information such as module loading and unloading behaviors.

Suspicious Instruction Shortlisting. From the execution states, our tool identifies instructions potentially vulnerable to DUI exploits. We use data dependency analysis to find the source of the memory address and the data used in memory access instructions. For a memory read operation, we also search forward to check whether the retrieved data is sent to other partitions through output interfaces. If the address is derived from the interface inputs and the data is derived or used in an attacker-controllable manner, we report this memory operation as a suspicious DUI vulnerability.

Dereference Behavior Analysis. Our tool generates the symbolic formula, called the *access formula*, to capture all the constraints from the interface inputs to the suspicious instruction. Then it analyzes the access formula to verify the DUI vulnerability and to assess the capability of attackers in controlling the memory space of the vulnerable partition.

DUI Detector reports the verified DUI vulnerabilities together with their severity to developers, helping them to fix the vulnerable code. Next, we introduce the key phases of the DUI Detector.

3.2 Suspicious Instruction Shortlisting

From the collected execution states, we use data dependency analysis to track the data flow of the memory address and the data used in memory access instructions. The methods used to detect DUIs are given below.

To detect write DUIs, we check for the following conditions for each instruction. (1) It is a memory write instruction, i.e., instructions that write the data into memory, like `mov`, `add`, `push` and successful conditional move `cmov`. (2) The source operand

is derived from the interface inputs, or predictable to attackers. (3) The address of the destination operand is also derived from the interface inputs.

To detect read DUIs, it is insufficient to check just one single instruction. Other than the actual memory read operation, it is also necessary to identify the data flow from the read operation to output interfaces, as we discuss in Section 2. Hence, we use a two-step approach to identify a read DUI.

1. A memory read instruction is selected for further checking if it matches the following two conditions: (1) The instruction reads data from memory and saves the data into registers. Instructions reading from registers or without saving the data into registers are ignored. (2) The memory address is derived from the interface inputs.
2. For an instruction selected above, we perform forward slicing on the data flow of the destination operand (the retrieved data). If the data reaches an output interface, we report it as a potential read DUI.

Our tool generates a list of the suspicious instructions potential vulnerable to DUIs. However, strong constraints on the interface inputs could significantly limits the attackers' capability, even making the instruction unexploitable. Hence we need to analyze each suspicious instruction to confirm the vulnerability and assess attackers' capability.

3.3 Dereference Behavior Analysis

Given a suspicious instruction identified in the previous step, our tool extracts an access formula to represent the relationship between the interface inputs and the address or data used in the instruction. The access formula captures all the constraints in the execution states with respect to the interface inputs. There are four types of constraints in the access formula as follows.

- **Data-Flow Constraints.** Data-flow constraints describe the arithmetic relations between the address and the data in the DUI instruction and the interface inputs. They are presented as a sequence of arithmetic operations.
- **Control-Flow Constraints.** Control-flow constraints ensure that the attacked partition follows the same path as the one recorded in the execution states. We only consider the path constraints related to the interface inputs. Other path constraints are out of the attacker's control and are assumed to be satisfied.
- **Memory Space Constraints.** To reach the suspicious instruction, all the memory accesses should be legitimate. Specifically, the code must have the correct write or read permission of the accessed page. Otherwise, page fault exceptions will be raised and divert the execution path. This constraint limits the attacker's capability since only a subset of memory space is accessible.
- **Data Life-Cycle Constraints.** To create an effective attack, the malicious data (written data or retrieved data) must be used within its life-cycle. Otherwise, the suspicious instruction cannot be exploited. For example, if the malicious data written to a selected location is immediately overwritten by a benign value, the attack does not have any effect on the victim partition. To capture this constraint, our tool considers subsequent instructions in order to track the aliveness of the data.

The generated access formula captures all the constraints on the interface inputs to reach the suspicious instruction and continue the execution. By assessing the attacker's capability by exploiting the instruction, we can determine if the suspicious instruction is indeed a DUI vulnerability. If so, we report the suspicious instruction and the attackers' capability to developers.

Attackers' Capability Assessment. Attackers' capability is represented as the ability to control the address and the value in the memory operation. A larger memory range controllable by attackers indicates a stronger attackers' capability. However, due to the constraints on the interface inputs, not all the malicious inputs lead to a successful attack. The working inputs form a valid data space, and the attacker's capability is determined by the size of this space. Our tool constructs constraint queries to estimate this space size. Specifically, we assign concrete values or a memory range to the operands of the suspicious instruction, i.e., the address or the data. Then these assignments are added to the access formula as new constraints to form a query. By solving the query using a constraint solver, we get the answer to the following questions: **Q1:** Is there any input making the partition follow the same path to the suspicious instruction when the address or the data in the address have to be a given value? **Q2:** Is there any input making the partition follow the same path to the suspicious instruction when the address or the data have to be within a given range? **Q3:** Is it true that for any address or any data in the given range there is an input making the partition follow the same path to the suspicious instruction? The answer to the question **Q1** indicates attackers' capability on controlling specific addresses. This is useful to build real exploits, for example, writing the ROP gadget address to a function pointer. A negative answer to the question **Q2** helps filter out a memory range from attackers' capability. While a positive answer to the question **Q3** adds the queried range into attackers' capability.

We take several query strategies to efficiently answer these questions. These strategies are based on the bit-pattern analysis and the range analysis [33, 35]. Through these methods, we can estimate the valid memory space controllable by attackers for each of the suspicious instruction.

- **Initial Target Analysis.** We first consider the memory page permission to initialize the memory range. For a memory read operation, the target memory location has to be readable. For a memory write operation, the target memory location has to be writable. Using this method, the queried memory range is limited to the readable or writable memory space.
- **Bit-Pattern Analysis.** Bit pattern analysis uses queries that specify concrete values on particular bits (or all bits) of the target value [33, 35]. An example of the query is whether the last two bits of the address have to be 10. This gives the answer to question **Q1**.
- **Range Analysis.** The range query identifies whether the values inside a particular range are valid [35] or not. If all values are valid, we conclude that the queried range is a valid range. If no value is valid, we remove the range from the valid memory space. If only some values are valid, the query solver will give a concrete valid value. We use this value to divide the range to two subranges. Then we use the range query to query both subranges. This query answers question **Q2** and **Q3**.

Finally the report given by DUI Detector includes the identified DUI vulnerabilities, together with the attackers' capability obtained by exploiting such vulnerabilities. It points out all the vulnerable-prone code in the checked partition. This enables security analysts to focus their efforts on a particular portion of the code.

4 Implementation

We built a prototype of DUI Detector on a 32-bit Ubuntu 10.04 system by extending the BitBlaze [42] platform. The prototype uses STP [22] as the SMT solver to query the access formula.

4.1 Taint Propagation

DUI Detector uses taint analysis to track the data flow of the interface inputs. Data from the interfaces are bound with the taint information of the source. Taint information has two aspects: One aspect is the taint attribute, a flag indicating whether a particular memory byte is tainted or not. Another aspect is the taint record, which contains the sources of the taint attribute. We use TEMU, the dynamic analysis engine of BitBlaze, as the base of taint propagation. However, there are several problems when we use TEMU to build our tool. Next we discuss these problems and present our solutions.

Finer-Gained Taint Record Propagation. Since we need to capture all the execution constraints, the taint propagation has to be accurate to permit the identification of all data sources. The normal taint propagation focuses on taint attribute propagation, and pays less attention on the taint record propagation. For example, for a given instruction, TEMU checks all its operands, and copies the taint records of the first tainted operand to the destination operand. This propagation method loses some taint sources. For example, in the instruction `add %ebx, %eax`, if `eax` and `ebx` are tainted by different data sources, the taint sources of `ebx` get lost. To solve this problem, we instead identify all the tainted source operands and copy all distinct taint records to the destination operand. As a result, the taint records for each operand capture all the data sources used to derive the operand.

1-Level Table Lookup. If a memory read address is tainted, taint analysis has to decide to propagate the taint to the destination operand or not. Table lookup is a method to propagate the taint. However, table lookup results in over-tainting problem, leading to a high false positive. A better tainting method for table lookup is necessary to capture the read DUI and avoid the over-tainting problem. We observe that as more table lookups are performed, attackers likely have increasingly less influence over the destination. As such, we propose the 1-level table lookup: Only propagating the taint for a single level of memory indexing. When the tainted data retrieved from table lookup are used as an index again, we will not propagate the taint. Our implementation uses the most significant bit of the taint attribute to indicate whether it is tainted through table lookup or not. Note that 1-level table lookup will miss attacks that utilize high-level table lookup to corrupt memory locations. However, we believe that the benefit on false positive reduction overweighs the false negative introduced since attacks with high-level lookup are rare in real-world attacks. With 1-level table lookup, our tool captures memory read

operations that are strongly controlled by attackers, and skips the weakly-controlled operations.

Taint Propagation for XMM registers. XMM (eXtended Multi-Media) registers are used to speed up the memory operation (e.g., `memcpy`), by joining several 4-byte copies into a single 16-byte operation. TEMU does not support the taint propagation through XMM registers. When tainted data are copied into an XMM register, the taint information gets lost. To support the taint propagation, we extend TEMU to correctly propagate taint to XMM registers and read taint information from XMM registers.

4.2 Access Formula Generation

We use VINE [42], the static analysis component of BitBlaze, to generate the formula for memory access from the trace. As discussed in Section 3.3, there are four types of constraints affecting memory access. However, VINE only generates two constraints, the data-flow constraint and the control-flow constraint. To bridge this gap, we develop tools to add additional two constraints into the formula. There are two steps to generate the memory space constraints.

1. In the guest OS, we insert a kernel module to detect the module loading and unloading behaviors. The kernel module sends the update information of the loaded and unloaded module to TEMU. We log such information together with the number of traced instructions when a behavior happens.
2. Using the log file, we can construct the readable and writable memory ranges for each instruction. Specifically, we collect all the modules that are still loaded in the memory for a given instruction. The union of their readable and writable memory ranges is the valid memory space. We add the memory range as a memory space constraint to the access formula.

To generate the data life-cycle constraints for a particular instruction, we search forward from the given instruction in the trace to find the first memory write instruction that overwrites the data at the same address. We call this instruction the update instruction. The data life-cycle of the data starts from the given instruction, and ends at its update instruction.

5 Evaluation

We evaluated our approach in the following system: The host OS is a 32bit Ubuntu 10.04 system, running on Openstack Cloud with two 2.4GHz vCPUs and 4GB RAM. The guest OS in TEMU is a 32bit Ubuntu 9.10 system. Next, we present our evaluation results and then discuss the security implication of our findings.

5.1 Efficacy

We applied DUI Detector on privilege-isolated programs to detect DUI vulnerabilities in protected partitions. We focus on two particular isolation schemes: The isolation

between malicious OS kernels and user space programs [17, 25, 32, 41, 48], and the isolation between malicious libraries and main programs [19, 21, 44–46]. We ran several programs on Linux system to get the execution trace, which were written to drive the execution through communications between different partitions. DUI Detector successfully detected read DUI and write DUI vulnerabilities in the protected user space code and the protected program main code. Further, DUI Detector assesses the attackers' capability obtained by exploiting such vulnerabilities. Next we present the details of these DUI vulnerabilities.

User-Kernel Isolation. A few proposals remove the OS kernel from the trusted base of the program execution, like hardware-based isolation (e.g., Flicker [31]) and hypervisor-based isolation (e.g., Overshadow [17]). These isolation schemes are designed to protect the sensitive data in user-space programs from the malicious kernel, so the kernel have no direct access to program memory space. Our goal is to detect DUI vulnerabilities inside protected user-space programs that allow the malicious kernel to corrupt programs' private user-space memory.

— Glibc code exploitable by `brk`**.** A write DUI was detected in the `malloc` function, which manages the heap memory for programs. The `malloc` calls the `brk` system call to request a new heap memory and takes the return value as the break value (the upper bound of data segment). Before looking into the detected DUI, we first illustrate the logic in `malloc` handling the return values of `brk`.

```
1  addr1 = brk(0);                  // get the current brk value
2  addr2 = brk(argument);           // request more space
3  *(addr1 + 4) = addr2 - addr1;    // store the size as metadata
```

This code snippet calls `brk` twice to create a heap memory region. The first `brk` call on line 1 is used to get the current break value (saved in $addr1$), which is the start address of the heap. The second `brk` call on line 2 is used to request more memory space and the new break value is stored in $addr2$. The memory location $addr1 + 4$ is used to store the size of the allocated data chunk, which is $addr2 - addr1$ in this case. The code on line 3 stores the size value into the metadata address. One of our tested programs invokes the `malloc` library call to call `brk`. In the recorded execution states, we found two instructions that match the write DUI Pattern 1, as listed below.

```
1  mov %eax, 0x4(%edx)
2  ...
3  mov %eax, 0x4(%edi)
```

For each instruction, both the value and the memory address are derived from the return values of `brk` system calls. By manipulating the system call return values, the malicious kernel can write any value into an arbitrary address in the victim process, even if the process is protected by encryption. We analyzed the capability of attackers and found that only the second instruction is exploitable. For the `mov` instruction on line 1, the data life-cycle constraints show that the value is immediately overwritten by another benign value. For the instruction on line 2, the first return value has to be a

multiple of 8. We show the constraints on the return value generated by our tool below, where the $brkn$ is the nth return value. DUI Detector generated the payloads in order to exploit this DUI vulnerability. The generated payloads successfully wrote the given address to the selected stack address.

```
1  condition( brk1%8 == 0 && brk2>brk1 )
2  address = brk1 + 0x2718;
3  data    = (brk2 - brk1 - 0x2718) | 0x1;
```

To explore other paths, we changed the condition to invalidate one of the constraints. The following are two conditions that lead to the write DUIs in other paths. The last one is the scenario of the Iago attack [15]. Note that DUI Detector accurately identified the constraints of Iago attack: The address has to be non-multiple of 8 and the data write to the memory has to be congruent to 1 modulo 8.

```
1  condition(brk1%8 != 0 && brk1<brk2<brk3)
2  address : relies on brk1;
3  data    : relies on brk1 and brk2;
```

```
1  condition(brk1%8 != 0 && brk1<brk2>brk3)
2  address : relies on brk1;
3  data    : relies on brk1 and brk3;
```

— *Glibc code exploitable by* `mmap2`. The second DUI vulnerability in Glibc is in the code handling the `mmap2` system call. The `mmap2` system call on Linux is used to map files or devices into memory in the Linux system. It is widely used by programs to map large files into memory. From the execution trace, we identified a total of 1,653 suspicious instructions matching write DUI patterns. We further reduced them to 302 based on the attackers' capability analysis. Analysis of the remaining 302 instructions reveals that all of them use values derived from the first 3 `mmap2` return values. Here we show the very first instruction among them. This is a write DUI, where the memory address and the data are derived from the first and the third `mmap2` return values.

```
1  mov %eax, 0x1ac(%edi)
```

Using the queries we discuss in Section 3.3, we identified the valid memory space which the attacker can write values to. For a stack memory range over `0x0BFFF000` to `0x0BFFF2FF`, we found that the attacker has no control over addresses whose last four bits are 1100 or 0100.

— `cat` *exploitable by* `read` *and* `write`. The UNIX utility program `cat` reads data from the given files, concatenates the content and writes them out to the standard output file. This behavior results in consecutive file read and write operations. The `cat` program we used is a derivative of the BSD `cat` program[1]. We identified read DUIs in the `cat` code, which can be exploited by malicious kernel to steal program's private information. To illustrate the read DUI, we present the pseudo code below.

[1] `http://www.opensource.apple.com/source/text_cmds/text_cmds-87/cat/cat.c`

```
1  nr = read(rfd, buf, size);
2  for(off = 0; nr; nr -= nw, off += nw)
3  {
4    nw = write(wfd, buf + off, nr);
5    if (nw == 0 || nw == -1)
6      goto error;
7  }
```

The loop condition nr is fully controlled by the malicious kernel. First, it is initialized by the return value of the `read` system call on line 1. For each loop, it is updated by the return value `nw` of the `write` call on line 2. `nw` is also used to advance the buffer for the next `write` call. When the kernel is changed to be untrusted, isolation mechanisms use deep copy to duplicate all system call parameters to a shared memory between kernel and process [41]. In this case, by manipulating the return value `nw`, the malicious kernel drives the process to copy its private data into shared memory space. With further capability analysis, we find that the attacker has full control over the value, i.e., the attacker is able to access any memory with the values ranging from `0x00000000` to `0xFFFFFFFF`.

Library Isolation. Dynamic shared libraries are linked to software process at the runtime. Since the dynamic library lives in the same memory space with the program's main code, any vulnerability in the library is inherited by the program. Memory separation designs [44–46] provide transparent memory isolation between the main code and libraries. The goal is to prevent the untrusted libraries from directly accessing the main memory. However, attackers can still leverage the DUIs in the main code to indirectly access the main memory.

— *Programs Using* `libsdl`. The Simple DirectMedia Layer (SDL) library provides programming interfaces to access low lever hardware, like keyboard, screen, audio and so on. The main program requests an SDL object and performs operation through the SDL object. For example, the main program can request a screen object, and then invoke the screen object methods to set display attributes, like colors and fonts. When the library isolation technique Codejail [45] is used, the SDL library code cannot directly access the memory of the main code. A monitor module will selectively commit the memory changes from the library to the main code. However, the memory isolation provided by Codejail cannot prevent the memory access from the library to the main memory through DUIs in the main code. We write a simple program that requests a screen object from the SDL library and then sets the color attribute. The pseudo code of the simple program is shown below.

```
1  screen = SDL_SetVideoMode(...);  // get framebuffer surface
2  color = SDL_MapRGB(...);         // get a pixel value
3  pixmem16 = screen->pixels + x + y * pixelsperline ;
4                                   // get pixel address
5  *pixmem16 = color;               // set the color
```

Our tool detected the write DUI in the main code (on line 5) of this simple program. A malicious SDL library can exploit this DUI vulnerability to corrupt any memory location of the main code, even if the main program is protected by memory isolation schemes. Using attackers' capability estimation, our tool reports that there is no limitation on the address or value, which means that attackers have full control of the main code memory through the DUI exploit.

5.2 Performance

Table 1. Performance of DUI Detector. T1 is the time for trace generation; T2 is the time to get the access formula; T3 is the time to solve the formula. "Inst. #" is the number of executed instruction, while "Infl. #" is the number of tainted instructions. All times are measured in second.

Trusted Part	Untrusted Part	APIs	DUI	Inst. #	Infl. #	T1	T2	T3
user space	Linux kernel	brk	write	168,089	103	21.79	1.70	0.18
user space	Linux kernel	mmap2	write	167,644	69,486	21.19	2.94	3.11
cat code	Linux kernel	read, write	read	2,288,914	684	104.76	16.58	0.16
main code	SDL library	SDL APIs	write	100,424,507	68	7574.23	1.52	0.10

Table 1 shows the performance details of each experiment conducted using our tool. We can see that our tool is able to analyze and detect a DUI vulnerability in a few minutes. The time required for the generation of the trace is largely dependent on the number of instructions that are generated and logged in the trace. On the other hand, the amount of time required for the generation of the STP formula is very small. For the STP formula solving, the time required highly varies due to its dependence on the query inputs, formula and how quickly the STP solver can obtain a solution for us.

5.3 Security Implications

Our tool detected DUI vulnerabilities in different program transformation scenarios, including untrusted kernel isolation and untrusted libraries isolation. In this part, we discuss the security implications of our findings.

- *Simple memory isolation is inadequate to prevent unauthorized memory access.* Although a lot of designs aim to prevent the malicious partition from accessing the protected memory, our result shows that simple memory isolation cannot completely stop the unauthorized memory access. DUI vulnerabilities inside the protected partition still allow other partitions to access arbitrary protected memory.
- *API-review is necessary to provide a secure environment*. Since DUI vulnerabilities can be leveraged to mount attacks through interfaces, developers need to pay special attention to the checking code on interface inputs when the legacy code is retrofitted into a memory isolation model. More specifically, there is a need to review the interfaces between trusted and untrusted partitions.

6 Discussion

In this section we present the limitation of our work and discuss the possible defense against the DUI exploits.

Code Coverage. Our analysis only considers one particular code path executed during the trace generation. However, it is possible for the program to have other DUI vulnerabilities in other paths. We employ an iterative process to detect other DUI vulnerabilities. Specifically, after the analysis for one execution path, we invalidate the path condition in the control-flow constraints and require the solver to provide an input that satisfies the invalidated condition [23]. The given input makes the program follow a new code path. The same analysis is performed on it and this process is repeated until no additional new path can be generated. This may lead to the path explosion problem [1]. To mitigate the problem, we only invalidate the conditional branches that are affected by untrusted input to generate the new path. Existing methods to mitigate path explosion, like [30, 43] can also be considered.

Defense. Once a DUI vulnerability has been identified, developers can mitigate the consequences of the vulnerability by introducing proper checks to the vulnerable code. Different checks should be used accordingly based on the type of the interface inputs. For the Glibc `brk` attack, where the interface inputs are addresses, the sanitization code needs to make sure that the returned address either equals to the requested one or points to a newly allocated memory region. For operation counters (e.g., the return value of the `read` system call), sanitization code should perform strict checks on the length, like comparing it with the file size or the buffer size.

7 Related Works

Vulnerability Detection. Symbolic execution and dynamic taint analysis are two techniques that are commonly used for vulnerability detection. In symbolic execution, the program is executed with symbols rather than concrete values. Operations on the inputs are represented as an expression of the symbols, naturally providing constraints on possible values of the input after each operation. As a result, symbolic execution [28] has been extensively used in program testing and vulnerability analysis [8, 12, 13, 34, 40]. Dynamic taint analysis is another technique frequently used to detect vulnerabilities. In taint analysis, attacker-controlled inputs are usually marked with a tag. This tag is propagated whenever the data is derived from the input. This enables the analyst to determine the data flow and the attackers' influence. A series of work has utilized taint analysis to detect and analyze vulnerabilities [9, 14, 49] and malware [18, 47]. Newsome *et al.* [36] proposed using dynamic taint analysis to find bugs. In these methods, attacks are detected when the tainted data are used in a dangerous way, like jump address or system call parameters. Our approach differs in application of these techniques. In order to detect DUI vulnerabilities, our focus is on detecting certain access pattern while at the same time considering implicit constraints such as memory constraints. As such, our approach aims to obtain a better understanding of the vulnerability in addition to detection.

Automatic Exploit Generation. The goal of the automatic exploit generation is to generate a working payload that successfully compromises the vulnerable program. Heelan *et al.* [24] discussed algorithms to automatically generate exploits to hijack the control flow for a given vulnerable path. Brumley *et al.* [10] proposed the automatic patch-based exploit generation for a given vulnerable program together with security patches. A followup work [2] presents an automatic exploit generation tool for buffer overflow and format string vulnerabilities. Felmetsger *et al.* [20] proposed AEG on web applications. In another work [26] we present an automatic method to generate data-oriented attacks. Different from these AEG-style approaches, our goal is to estimate the attackers' capability. Hence, rather than obtaining the payload for a vulnerable program, we quantify the potential severity of the vulnerability.

Privilege Separation in Software Systems. Privilege separation is a way to realize the principle of least privilege in software designs. It is often achieved by using memory isolation to protect resources of high-privileged partitions from low-privileged ones. For examples, Provos [39] retrofitted OpenSSH with a privilege-separated design and other methods [6, 11, 27] automatically separate and isolate components within monolithic legacy programs. Other security solutions proposed new threat models. For example, some [17, 29, 31] treat the kernel as potentially untrusted and remove it from the trusted computing base. However, the work [15, 38] shows that just isolating the components is insufficient as attackers might be able to leverage on poorly designed legacy interfaces to compromise the isolated components. Our solution complements this work with a systematic method to detect DUI vulnerabilities when adopting new isolation schemes.

8 Conclusion

In this paper, we present a systematic solution to detect arbitrary memory access vulnerability in binary programs. Our approach builds access formula for a binary using program analysis techniques. The formula is then utilized to detect the memory access patterns that can be leveraged by attackers to perform arbitrary memory accesses. Detailed analysis is also performed to assess the capability of attackers using such vulnerabilities. We demonstrate the effectiveness and accuracy of our approach in the evaluation, where we present four case studies of DUI vulnerabilities in programs utilizing isolation schemes. Finally, we provide the security implications based on the results of the evaluation.

Acknowledgments. We thank Xinshu Dong and the anonymous reviewers for their insightful comments. This research is supported in part by the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate, and by a research grant from Symantec.

References

1. S. Anand, P. Godefroid, and N. Tillmann. Demand-driven Compositional Symbolic Execution. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.
2. T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: Automatic Exploit Generation. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, 2011.
3. A. Barth, C. Jackson, C. Reis, and T. G. C. Team. The Security Architecture of the Chromium Browser. Technical report, 2008.
4. D. J. Bernstein. Some Thoughts on Security After Ten Years of Qmail 1.0. In *Proceedings of the 14th ACM Workshop on Computer Security Architecture*, 2007.
5. S. Bhatkar, D. C. Duvarney, and R. Sekar. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proceedings of the 12th USENIX Security Symposium*, 2003.
6. A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, 2008.
7. R. Blum. *Postfix*. Sams, Indianapolis, IN, USA, 2001.
8. D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song. Towards Automatic Discovery of Deviations in Binary Implementations with Applications to Error Detection and Fingerprint Generation. In *Proceedings of 16th USENIX Security Symposium*, 2007.
9. D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards Automatic Generation of Vulnerability-Based Signatures. In *Proceedings of the 27th IEEE Symposium on Security and Privacy*, 2006.
10. D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, 2008.
11. D. Brumley and D. Song. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *Proceedings of the 13th USENIX Security Symposium*, 2004.
12. C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, 2008.
13. C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, 2006.
14. D. Caselden, A. Bazhanyuk, M. Payer, S. McCamant, and D. Song. HI-CFG: Construction by Binary Analysis, and Application to Attack Polymorphism. In *Proceedgins of 18th European Symposium on Research in Computer Security*, 2013.
15. S. Checkoway and H. Shacham. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.
16. S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data Attacks Are Realistic Threats. In *Proceedings of the 14th USENIX Security Symposium*, 2005.
17. X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: A Virtualization-based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.
18. M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic Spyware Analysis. In *Proceedings of USENIX Annual Technical Conference*, 2007.

19. U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software Guards for System Address Spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006.
20. V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. Toward Automated Detection of Logic Vulnerabilities in Web Applications. In *Proceedings of the 19th USENIX Security Symposium*, 2010.
21. B. Ford and R. Cox. Vx32: Lightweight User-level Sandboxing on the x86. In *Proceedings of USENIX Annual Technical Conference*, 2008.
22. V. Ganesh and D. L. Dill. A Decision Procedure for Bit-vectors and Arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification*, 2007.
23. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proceedings of the 26th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.
24. S. Heelan. Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities. Technical report, Computing Laboratory, University of Oxford, September 2009.
25. O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. InkTag: Secure Applications on an Untrusted Operating System. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.
26. H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang. Automatic Generation of Data-Oriented Exploits. In *24th USENIX Security Symposium*, 2015.
27. D. Kilpatrick. Privman: A Library for Partitioning Applications. In *Proceedings of USENIX Annual Technical Conference*, 2003.
28. J. C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7), July 1976.
29. D. Lie, C. A. Thekkath, and M. Horowitz. Implementing an Untrusted Operating System on Trusted Hardware. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.
30. K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks. Directed Symbolic Execution. In *Proceedings of the 18th International Conference on Static Analysis*, 2011.
31. J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for Tcb Minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2008.
32. F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.
33. Z. Meng and G. Smith. Calculating Bounds on Information Leakage Using Two-bit Patterns. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*, 2011.
34. D. A. Molnar, D. Molnar, D. Wagner, and D. Wagner. Catchconv: Symbolic Execution and Run-Time Type Inference for Integer Conversion Errors. Technical report, UC Berkeley EECS, 2007.
35. J. Newsome, S. McCamant, and D. Song. Measuring Channel Capacity to Distinguish Undue Influence. In *Proceedings of the ACM SIGPLAN 4th Workshop on Programming Languages and Analysis for Security*, 2009.
36. J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, 2005.
37. PaX Team. PaX Address Space Layout Randomization (ASLR). `http://pax.grsecurity.net/docs/aslr.txt`, 2003.
38. D. R. K. Ports and T. Garfinkel. Towards Application Security on Untrusted Operating Systems. In *Proceedings of the 3rd Conference on Hot Topics in Security*, 2008.

39. N. Provos, M. Friedl, and P. Honeyman. Preventing Privilege Escalation. In *Proceedings of the 12th USENIX Security Symposium*, 2003.
40. D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani. Darwin: An Approach for Debugging Evolving Programs. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, 2009.
41. S. Shinde, S. Tople, D. Kathayat, and P. Saxena. PODARCH: Protecting Legacy Applications with a Purely Hardware TCB. Technical Report NUS-SL-TR-15-01, School of Computing, National University of Singapore, February 2015.
42. D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security*, 2008.
43. M. Staats and C. Păsăreanu. Parallel Symbolic Execution for Structural Test Generation. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, 2010.
44. R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM symposium on Operating systems principles*, 1993.
45. Y. Wu, S. Sathyanarayan, R. H. C. Yap, and Z. Liang. Codejail: Application-Transparent Isolation of Libraries with Tight Program Interactions. In *Proceedings of 17th European Symposium on Research in Computer Security*, 2012.
46. B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, 2009.
47. H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing System-Wide Information Flow for Malware Detection and Analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.
48. F. Zhang, J. Chen, H. Chen, and B. Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, 2011.
49. M. Zhang and H. Yin. AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications. In *Proceedings of the 21st Network and Distributed System Security Symposium*, 2014.