Run-time defense survey

Raphael Eikenberg
raphael@eikenberg.at

Abstract—**Protecting the integrity of user-space programs and the data they are processing plays an important role in software security. As increasingly sophisticated attacks are developed to circumvent security measures, research becomes more specialized and creative, too. For instance, research has focused on the defense of data-only attacks lately, as such attacks are becoming more relevant in practice. Moreover, there are tendencies toward active run-time measures for improving existing *control-flow integrity* (CFI) mechanisms and their enforcement. Also, recent studies have developed new methods to prevent temporal memory errors, especially *use after free* (UaF) exploits. This survey paper aims at providing an overview of the recent development in run-time defenses by presenting novel papers and comparing their approaches.**

1. Introduction

Run-time defenses are mechanisms in computer security that try to ensure secrecy and integrity of data of a running process as well as the availability of that process. That means they prevent information disclosure of sensitive data of a process and maintain control-flow integrity. While there are measures that can be taken to improve security at compile-time (e.g. PIE, RELRO), recent studies have focused on advancing security concepts for run-time. The reason for this is that run-time defense can be more effective regarding the dynamic methods that can be used during execution.

Modern exploits usually deal with memory corruption because they enable attackers to read, write or even execute memory in the worst case. Both – control data and non-control data – are currently targets of such exploits since they allow to undermine the control-flow integrity of the kernel or of a program.

Intentional memory corruption has been a problem for the last few decades. This is based on the weaknesses that come with low level programming languages like C and C++, which are still being used intensively for the development of large-scale user programs and commodity kernels nowadays. Even though techniques have been invented to make software written in such languages less error prone (e.g. SMEP, SMAP, ASLR), attackers are creative enough to find new ways to circumvent those measures, for example using *return-oriented programming* (ROP). This is why new defense strategies like *moving target defense* (MTD) and fine-grained *control-flow integrity* (CFI) gained a lot of interest in the last years. Both, MTD and advanced CFI methods play a role in the papers selected for this survey.

A notable advancement in this field concerns memory error detection. Google has created AdressSanitizer, a tool which uses shadow memory to detect various memory bugs including buffer overflows and is successfully used in projects like Firefox and Chromium. Also worth mentioning is how CFI is slowly becoming adopted in practice lately, where the *Reuse Attack Protector* (RAP) in particular is considered a pioneer because it also covers backward-edge CFI [1].

It is debatable whether memory error detectors or direct exploit mitigation techniques are better suitable to prevent intentional memory corruption. Both are valid approaches to the same problem and for now, both approaches are actively studied.

The papers selected for this survey were presented at the *USENIX Security Symposium*, the *ACM Conference on Computer and Communications Security*, the *ACM Workshop on Moving Target Defense* and the *NDSS Symposium*.

2. Research challenges

In the following section, we will have a detailed view at various papers that were recently published. This includes the work on two CFI designs PITTYPAT [2] and μCFI [3], as well as two methods to prevent temporal memory errors pSweeper [4] and Oscar [5] which will be discussed in section 2.1 and section 2.2 respectively. Furthermore, we will discuss the memory error detector MEDS [6] in section 2.3 and a technique for randomizing the virtual address space named Mixr [7] in section 2.4.

2.1. Control flow integrity

Control-flow integrity (CFI) is a broad term that describes the goal of computer security to keep the execution of the program within expected paths and therefore prevent an attacker from manipulating the execution path to their favor. *Forward-edge* CFI resticts the meaning to branches, jumps, calls or similar active modification of the program counter, while *backward-edge* CFI only deals with returns from such function calls [8].

Both, PITTYPAT [2] and μCFI [3] analyze existing solutions for CFI (e.g. π-CFI) and provide advanced techniques to further strengthen available enforcement methods.

Ding et al. [2] start off by explaining why such existing solutions do not suffice to protect the forward-edge control flow. They give a short code snippet to illustrate their reasoning, which can be seen in listing 1. As can be seen in the listing, a static CFI mechanism will determine two possible function pointers for `handler` in line 22, namely `priv` and `unpriv`, because depending on the user from which the request came from the `handler`

```
1  struct request {
2    int auth_user;
3    char args[100];
4  };
5
6  void dispatch() {
7    void (*handler)(struct request *) = 0;
8    struct request req;
9
10   while(1) {
11     // parse the next request
12     parse_request(&req);
13     if (req.auth_user == ADMIN) {
14       handler = priv;
15     } else {
16       handler = unpriv;
17       // NOTE. buffer overflow, which can overwrite
18       // the handler variable
19       strip_args(req.args);
20     }
21     // invoke the hanlder
22     handler(&req);
23   }
24 }
```

Listing 1. Code example used to introduce PITTYPAT [2]

will be assigned a different function. However, during run-time, only a specific function pointer would be valid when executing the instruction in line 22 since beforehand the conditional statement determines the value of `handler`. PITTYPAT is a CFI mechanism that takes advantage of this fact and tries to analyze the run-time behavior so that the number of valid targets decreases.

The set of all valid targets for an instruction is called its *points-to set*. The points-to set of a program is a mapping for all control-flow transfer instructions of that program to a set of targets. A control-flow transfer is regarded as valid only if the target is in that set for the instruction being executed. If a CFI mechanism detects a jump to a target that is not in this set at the according instruction, the program flow is corrupted.

Ding et al. [2] state that a previous study came up with a similar idea, called π-CFI. The problem with π-CFI is that once a target is considered as valid in a given context, the target will never be considered as invalid any more, even after leaving the context of execution. In other words, validations will always be persistent. As a consequence, when considering the case of listing 1, once a request of an admin is processed, `priv` becomes a valid target. Upon the next iteration of the while loop, `priv` is an element of the points-to set for the instruction in line 22, regardless of the possibility that a request of a normal user is handled currently.

PITTYPAT, however, is more dynamic in that sense and calculates the points-to set each time the program executes the according instruction. Doing so must be done efficiently, as otherwise the performance overhead would make the use of the system impracticable. Ding et al. [2] decided to use an analysis module that keeps track of the points-to data and a kernel-space driver that gathers the process information needed to perform the analysis.

Prior to monitoring, the analysis module needs to be given a representation of the program to be able to infer the program state during execution. While the process is running, the driver collects run-time information using the *Intel Processor Trace* (Intel PT) and passes them on to the analyzer through a ring buffer. Once a system call is issued by the monitored program, the analyzer checks whether or not the taken control targets until then were valid and only permits the continuation of the program execution if that check was successful.

Two major challenges were identified by the authors during their work. One of them is that efficient tracking of the currently executed branch is difficult because Intel PT only provides very little information about which branch was taken. Manual processing has to be done to use that information. Ding et al. [2] concluded that it is enough to know the basic block that is currently being executed, as opposed to the specific instruction. The other challenge they faced during designing their defense mechanism was the implementation of an efficient analyzer. Known analyzers were either not precise enough or were not suited for use during run-time.

Hu et al. [3] on the other hand show how PITTYPAT could still be tricked, and suggest another design based on what they call the "Unique Code Target" (UCT) property. The UCT property requires the CFI mechanism to analyze the control flow and only regard a single next instruction as valid whenever a jump or branch happens. To show

```
1  typedef void (*FP)(char *);
2  void A(char *); void B(char *); void C(char *);
3  void D(char *); void E(char *);
4  // uid can be 0, 1, 2
5  void handleReq(int uid, char *input) {
6    FP arr[3] = {&A, &B, &C};
7    FP fpt = &D;
8    FP fun = NULL;
9    char buf[20];
10   if (uid < 0 || uid > 2) return;
11   if (uid == 0) {
12     fun = arr[0];
13   } else { // uid can be 1, 2
14     fun = arr[uid];
15   }
16   strcpy(buf, input); // stack buffer overflow
17   (*fun)(buf);
18   // fun is corrupted
19 }
```

Listing 2. Code example used to introduce μCFI [3]

why this property is needed, they provide another code snippet, which can be seen in listing 2. The difference to listing 1 is that the conditional variable is used to assign the correct function to `fun`. This means that no longer the knowledge of the basic block is sufficient to determine the single valid target, since in case the condition leads to the `else`-branch being executed, two different function pointers could be assigned. In fact, the points-to set still contains the valid target, but it includes other targets, too.

Again, just as with PITTYPAT, a monitor was used to collect data via Intel PT and an analyzer had to manage the points-to information. Since the run-time analysis of μCFI requires the knowledge of what is called "constraining data"[1], a technique was suggested to record such data during run-time in the executed process. The technique involves two functions, `write_data()` and `read_data()`. `write_data()` is integrated by the compiler wherever necessary in the monitored program to encode the constraining data and pass it to the Intel PT trace. `read_data()` is then used to retrieve that data from Intel PT and decode it in the monitor to allow for a more sophisticated approach of the points-to set. This

1. Data that can be used to determine the control-flow transfer target.

way, the execution context can be used from outside the process using hardware support.

As Hu et al. [3] state, the performance of decoding the data represented a big challenge because Intel PT only allows for densely-compressed data. Furthermore the difference between the analyzer-internal representation of the program and the actual binary-level path – resulting from compiler optimization – had to be kept in mind while implementing μCFI.

While Ding et al. [2] report that PITTYPAT introduces a run-time overhead of $12.7\,\%$ geometric mean using `SPEC CPU® 2006`, Hu et al. [3] measure an overhead of below $2\,\%$ for μCFI using the same method. According to the authors, that superiority in performance is likely due to the encoding used for passing information through Intel PT.

2.2. Temporal memory safety

Use after free (UaF) exploits make use of the fact that sometimes function pointers point to freed areas of the virtual memory. Such pointers enable attackers to run arbitarary code in the worst case.

The next two papers, pSweeper [4] and Oscar [5], aim to prevent UaF exploits using different approaches.

```
1  void (**someFuncPtr)() = malloc(sizeof(void*));
2  *someFuncPtr = &Elmo;   // At 0x05CADA
3  (*someFuncPtr)();       // Correct use.
4  void (**callback)();
5  callback = someFuncPtr;
6  ...
7  free(someFuncPtr);      // Free space.
8  userName = malloc(...); // Reallocate space.
9  ... // Overwrite with &Grouch at 0x05DEAD.
10 (*callback)();          // Use after free!
```

Listing 3. Code example used to introduce Oscar [5]

Dang et al. [5] give a good example of a situation where such protection would help, which is displayed in listing 3. As can be seen in the code example both `someFuncPtr` and `callback` point to the same function at first. After the space is freed, it is reallocated and could be filled with arbitrary code by an attacker. Then, the function is called again, which means that there is a risk of the program being exploited. Of course – in this example – the problem is clear, but this may not be the case for programs with complex structures which is why UaF exploits are dangerous in the first place.

To enforce temporal memory safety, Dang et al. [5] came up with an efficient memory allocation scheme. In fact, Oscar is an *implicit lock-and-key* mechanism meaning that each object is placed on its dedicated virtual page (called a *shadow page*), allowing for safe freeing of the page when the object is not needed any longer.

A *lock-and-key* mechanism is used to prevent unauthorized access to specific memory locations. Upon allocation a key is generated that is needed to access the memory location afterwards. While *explicit* lock-and-key mechanisms generate an actual key that is being stored separately and checked when accessing the memory, an *implicit* lock-and-key mechanism instead assigns each allocation its own virtual page that can be freed when the object is not needed any longer. Lock-and-key mechanisms work because other references to the same object would no longer be valid after `free()` was called, which prevents UaF to a certain extent, presuming that the virtual address space is never exhausted.

According to Dang et al. [5], Oscar consists of the following four elements.

- The allocation procedure of program memory has to be altered. When `malloc()` is called, additional memory needs to be reserved to save a reference of the canonical page.
- A reference to the canonical page has to be stored.
- System calls to manage shadow pages have to be added.
- The TLB pressure, i.e. how frequent accesses to different pages are.

To evaluate the overhead of their design, four implementation steps were added incrementally in order to point out the key slowdowns. After each step the overhead was measured using `SPEC CPU® 2006`. First, the allocation mechanism was changed so that `malloc()` uses `mmap(MAP_SHARED)` in the background which allows for aliasing memory pages. In the second step, the additional memory required for the aforementioned reference was taken care of. Next, the shadow pages were actually created, but not used by the program. Finally, in the last step the shadow pages were used. Altogether, those were the steps needed for a basic implicit lock-and-key scheme.

In short, their findings show that most of the measured benchmarks ran without significant overhead. However, specifically `mcf` and `milc` were slowed down by the first stage already which is due to the allocation mechanism introduced by Oscar. Also, the overhead of `gcc` as well as `astar` doubled in the fourth stage, which results from the usage and creation of the shadow pages according to the authors.

After the analysis of the overhead, Dang et al. [5] tried to lower the overhead by reducing the costs for managing shadow pages and using the `MAP_SHARED` mode of `mmap()`. One method consists of something called a *high water mark*, which is essentially a bound for virtual addresses requested so far, preventing reallocation of virtual pages while enabling the kernel to free the according metadata of the pages. Another improvement Dang et al. [5] came up with is the refreshing of shadow pages. Once a shadow page is freed, another shadow page is immediately created using *mremap()*, so that a whole system call can be saved. The third important change that was introduced to lower the overhead is the usage of `mmap(MAP_PRIVATE)` when possible. Overall, the more efficient version of Oscar could indeed improve the benchmarks of several measurements, including `gcc`, `mcf` and `milc`, while other measurements remained unchanged.

Finally, Oscar was made compatible with programs that use `fork()`. As was mentioned earlier, most pages are allocated using `MAP_SHARED`. To make children independent from their parents and to avoid crashes, those pages have to be copied over to new pages. Dang et al. [5] came up with the idea of using `mremap()` to do this.

A different approach to temporal memory safety is pSweeper [4]. According to Liu et al. [4], pSweeper is a run-time defense that neutralizes so-called dangling

pointers by looking for them in a separate thread. It is notable that the scanning of pSweeper can be split into multiple threads which leads to high scalability. The authors state themselves that such work has been done before, but they introduced new techniques that enable the design to increase in performance.

The techniques are called "Concurrent Pointer Sweeping" (CPW) and "Object Origin Tracking" (OOT). CPW is used to find dangling pointers while the underlying program is running by iteratively scanning through all live pointers. In essence, this does not yield better performance of the scanning itself, but since the procedure is outsourced to another thread, the program itself can run faster. OOT on the other hand, is a way to encode debugging information into a dangling pointer that causes a crash. When cleaning a dangling pointer, pSweeper does not write its value to a defined constant, but uses the opportunity to write its value by encoding information that help identifying the object related to the pointer.

Since UaF could still be exploited in a time frame between a free and the cleaning by pSweeper, it was designed so that frees always happen after the scanning routine ends. This means that frees are delayed so that UaF cannot be exploited. A free happening during that routine is postponed until the end of the next routine.

To identify pointers during run-time, pSweeper uses a compile-time mechanism to statically index all relevant addresses. To overcome race conditions on the stack, all pointers on the stack are moved to a separate stack, while complex data structures are stored on the heap instead. This way, pSweeper prevents false-positives when scanning for dangling pointers. For pointers on the heap, live monitoring of assignments during execution has to be deployed. As opposed to the previous work on this topic, Liu et al. [4] decided not to do the full tracking synchronously, but to bookmark the affected addresses for later evaluation.

Another major challenge beside improving speed was the prevention of propagation of dangling pointers. Liu et al. [4] introduced a scheme to their design so that variables are always checked for such propagation after an assignment.

Lastly, the overhead of pSweeper was measured using `SPEC CPU® 2006`. The rates at which the pointers were scanned were varied between $1\,\mathrm{s}$, $500\,\mathrm{ms}$ and $0\,\mathrm{s}$ (i.e. without pause in between) during the measurements. The average run-time overhead of pSweeper was $17.2\,\%$, $13.9\,\%$ and $12.5\,\%$ respectively, while Oscar imposed $40\,\%$. Still, especially `perlbench` and `xalancbmk` ran with an overhead of above $70\,\%$ for all rates.

Interestingly, higher sweeping intervals yielded higher overhead. According to Liu et al. [4], this is due to the fact that higher intervals imply more time spent when allocating new memory since the already freed memory can be reused only when pSweeper finished the current scan.

Considering the memory overhead, Oscar has only $52\,\%$ according to the measurements, which clearly outperformed pSweeper's overhead of roughly $112.5\,\%$.

2.3. Memory error detection

Memory errors can be divided into two categories, *temporal memory errors* and *spatial memory errors*. Temporal memory errors are those where freed memory is dereferenced or where uninitilized pointers are used. In contrast, spatial memory errors occur when data is written out of bounds in memory to locations that are not intended to be used for that data.

MEDS [6] is a modern memory error detector used to terminate programs that interact with the memory in a way they should not. Han et al. [6] aim at improving existing approaches using what they call an "infinite gap" and an "infinite heap". This allowed them to improve the detection capability compared to the existing standard solution, AddressSanitizer by Google.

Han et al. [6] based their design on *redzone-based detection* instead of *pointer-based detection* to support better compatibility with programs written in C/C++. While the latter tracks the capabilities of pointers and checks them on access, the *redzone-based detection* tries to prevent faulty access by inserting gaps in between memory allocations to handle spatial errors. Whenever an access to such a gap is registered, the program is forced to terminate as the behavior was faulty. To prevent temporal errors, the allocated blocks are tried to never be reused after they were freed for the first time.

In order to archive theses protections, MEDS comes with its own memory allocator named MEDSALLOC. MEDSALLOC makes use of the whole virtual address space on 64-bit systems allowing for both larger gaps between two memory allocations and unlikelier reuse of addresses. For *redzone-based detection*, large gaps are important because if a faulty memory access overlaps with another object allocated on the heap, the error will not be detected as such. On top of that, Han et al. [6] made use of page aliasing, so that physical memory usage is kept at a minimum.

Compared to AddressSanitizer, Han et al. [6] argue that their design benefits from the "infinite gap" and the "infinite heap". Specifically, Han et al. [6] criticize the small gaps AddressSanitizer uses, which only range from $16\,\mathrm{B}$ and $2048\,\mathrm{B}$. Theoretically, the gaps between two objects would have to be infinite to be able to detect all spatial memory errors, and the heap would have to be infinite so that no address is ever reused. Of course, this cannot be implemented in a real-world scenario, but MEDS tries to mimic such properties by efficiently handling virtual memory.

On the one hand, the allocator MEDSALLOC always tries to allocate memory at addresses that have not been used before. Since MEDS makes use of the whole virtual address space to do so, plenty of allocations can be made until an address must be used again. On the other hand, MEDS tries to be memory efficient by sharing physical frames for multiple virtual pages. Since page permissions always affect a whole page, only a single object is placed per page. However, to reduce the memory overhead introduced by that, multiple virtual pages will use the same physical frame in order to store the data. This is possible because usually objects will not take up a whole page on their own.

Because AddressSanitizer's management of redzones was too inefficient for the requirements of MEDS, a new scheme was introduced. MEDS is aware of two different types of redzones. The first type is a redzone that includes whole pages, which means that no detailed zone information is needed for such a page, since an access always represents faulty behavior. For the first type, MEDS does not allocate any physical memory, because if the page is not mapped at all, access will lead to a page fault anyways. The other type is a redzone that shares the page with program data. This type needs more granular management and is realized as *shadow memory*. The *shadow memory* is a bit-vector that contains the information needed to decide whether a byte of a page is either a redzone or program data.

Han et al. [6] further claim that the security guarantees of MEDS are stronger than those of ASLR, which is why the design of MEDS deals with memory in a contiguous way.

According to Han et al. [6], their design is already very mature and could be used in production. The implementation was realized as a LLVM module and provides necessary routines for allocation and deallocation. As can be read in their publication, the design already works for major web servers and clients.

Moreover, Han et al. [6] evaluated the overhead of their design using various benchmarks. They state that MEDS enhances memory protection not only because it is more precise than Google's AddressSanitizer, but also because it takes MEDS less time detect errors. Still, the performance makes it impracticable to use MEDS in production whenever performance plays a role. Benchmarks show how MEDS introduces a $44\,\%$ overhead for Chrome and a $250\,\%$ overhead for Nginx, while AddressSanitizer performed with $17\,\%$ and $10\,\%$ overhead respectively. Also they reported a memory overhead of $301\,\%$ when running Nginx.

2.4. Moving target defense

A *moving target defense* (MTD) mechanism is a protection technique where the attack surface is changed throughout time. This means the possible attack vectors are dynamic over time, whereas for classical protection mechanisms they are static.

MTDs are especially helpful against ROP attacks. *Return-oriented programming* (ROP) is an attack technique used to circumvent standard defense mechanisms by exploiting code that is already part of the running program. This is sometimes also referred to as *code-reuse*. MTDs can efficiently prevent ROP attacks because they can make all the gathered knowledge of the attacker unusable and thus thwart the attack. A *ROP gadget* is a piece of code that is being exploited during a ROP attack.

Mixr [7] is an MTD system which is being referred to by the authors of the according paper as a "run-time ASLR". It allows for frequent and granular rerandomization of the program space during the execution of the program. This helps preventing information leakage and fights the weaknesses of normal ASLR. Especially worth mentioning is that no source code or other additional information is needed in order to apply Mixr to programs, and the kernel can be left unmodified, too. To the knowledge of Hawkins et al. [7], Mixr is the first system with the capabilities of randomizing the virtual memory of a program without such additional requirements while meeting the same level of security.

The goal of Mixr is to change the behaviour of the program that is being secured so that it randomizes blocks in its virtual memory during run-time. The behaviour of the program is modified in its binary form. To rewrite a program, Mixr makes use of Zipr, which is a tool for program transformations.[2] Zipr offers an API which can be used to split a program into so-called *dollops*. A dollop is a piece of original code that is handled by Mixr like an atomic block of code, meaning that its instructions will never be taken apart by the randomizer. That means when the execution of a dollop has finished, the next dollop must be executed in order to maintain the original program flow.

In the best case, a dollop ends with an instruction performing control flow transfer, as this can be used by Mixr to jump to the next dollop. If this is not the case, however, Mixr adds an additional jump to the end of the dollop to perform this task.

Since dollops will be moved around in the virtual memory, additional data is required for each dollop to handle relative memory access and others. To be more precise, a *read/write table* (RW table) and a *dollop entry table* (DE table) are stored for each dollop. The former provides the information to perform relative memory access, while the latter provides similar information for control flow transfer. The metadata associated with a dollop together with the dollop itself is called a "dollop bundle". A dollop will only be moved together with its metadata, which means that Mixr essentially randomizes the positions of dollop bundles.

To keep track of the positions of dollop bundles, a dollop table is used. The position of the dollop table is not randomized during the execution of the program, as according to Hawkins et al. [7] the data contained in the table does not have to be kept secure.

When rewriting an executable using Mixr, the user can decide how often a rerandomization occurs, when it occurs and which algorithm is applied to perform the randomization. Also, the size of dollops can be adjusted by the user. All available algorithms for randomization have in common that they select a pair of dollop bundles that will be swapped.

What differentiates Mixr from other approaches is that the randomized blocks – which are called dollop bundles in this case – are more flexible in terms of size. In contrast to the adjustable size in Mixr, other approaches leave the size and shape of the randomized blocks to the output of the compiler. Since that leads to more ROP gadgets per block Hawkins et al. [7] argue their research provides additional security over existing solutions.

Lastly, to evaluate the overhead introduced by Mixr, `SPEC CPU® 2006` was used to create benchmarks. In total there were three different sources of overhead determined. The first overhead measured was the growth in size of the program binary and was found to be roughly $351\,\%$. The two other main sources were the run-time overhead of the separation of the program into dollops and the run-

2. Hawkins et al. [7] were involved in the design process of Zipr.

time overhead of the randomization itself. Together, they introduced an overhead of 225 %.

3. Future challenges

A big issue with most of the solutions presented in this paper is that their usage is limited to a trade-off in performance. Since software security concerns not only environments with high security standards, one has to be able to weigh between performance and security. The approaches discussed in section 2 definitely demonstrate that usable security mechanisms exist. However, there is still work to be done to be able to make use of those mechanisms in regards to memory and run-time overhead.

Specifically Mixr has a few weaknesses that are to be fixed yet. For instance, Hawkins et al. [7] intentionally left out the support for multithreaded programs in their work. Also, they mention that at a later point their algorithm could rerandomize the randomization procedures in virtual memory, too.

In other works, ideas for improvement were suggested as well. For example, μCFI lacks the support for signal handling and shared libraries which could be added. Moreover, Han et al. [6] claim their solution could improve in performance by adding functionality to the Linux kernel.

In the next years, research will be challenged with the question of whether or not memory error detectors can replace direct exploit mitigation techniques in the long run. On the one hand, their ability to improve security is said be be superior. On the other hand, their overhead makes it very hard to deploy them in practice. [6]

4. Conclusion

In this paper, we have discussed modern approaches to the defense mechanisms in software security.

First, two different CFI designs – namely PITTYPAT [2] and μCFI [3] – were presented and compared. Both solutions show how recently added hardware features can be used to improve the security of a system.

Next, we had a look at the current challenges with temporal memory errors and how they can be prevented. pSweeper [4] as well as Oscar [5] showed how to mitigate UaF exploits using different techniques.

After that, the memory error detector MEDS [6] was introduced. Han et al. [6] faced the challenge of improving the state-of-the-art detector AddressSanitizer.

Lastly, a randomizer for the virtual address space called Mixr [7] was presented. While its overhead renders the system likely not usable in practice, the work shows how MTD systems keep up with the rapid development of modern attacks.

References

[1] Open Source Security, Inc, "RAP™ Demonstrates World-First Fully CFI-Hardened OS Kernel," 2017, (accessed 16-December-2018). [Online]. Available: https://grsecurity.net/rap_announce_ret.php

[2] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee, "Efficient protection of path-sensitive control security," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC, Canada: USENIX Association, 2017, pp. 131–148. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/ding

[3] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee, "Enforcing unique code target property for control-flow integrity," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: ACM, 2018, pp. 1470–1486. [Online]. Available: http://doi.acm.org/10.1145/3243734.3243797

[4] D. Liu, M. Zhang, and H. Wang, "A robust and efficient defense against use-after-free exploits via concurrent pointer sweeping," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: ACM, 2018, pp. 1635–1648. [Online]. Available: http://doi.acm.org/10.1145/3243734.3243826

[5] T. H. Dang, P. Maniatis, and D. Wagner, "Oscar: A practical page-permissions-based scheme for thwarting dangling pointers," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC, Canada: USENIX Association, 2017, pp. 815–832. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/dang

[6] W. Han, B. Joe, B. Lee, C. Song, and I. Shin, "Enhancing memory error detection for large-scale applications and fuzz testing," 01 2018.

[7] W. Hawkins, A. Nguyen-Tuong, J. D. Hiser, M. Co, and J. W. Davidson, "Mixr: Flexible runtime rerandomization for binaries," in *Proceedings of the 2017 Workshop on Moving Target Defense*, ser. MTD '17. New York, NY, USA: ACM, 2017, pp. 27–37. [Online]. Available: http://doi.acm.org/10.1145/3140549.3140551

[8] The Clang Team, "Control Flow Integrity Design Documentation," 2018, (accessed 16-December-2018). [Online]. Available: https://clang.llvm.org/docs/ControlFlowIntegrityDesign.html