iCruiser: Protecting Kernel Link-Based Data Structures with Secure Canary

Li Wang, Dinghao Wu, and Peng Liu
College of Information Sciences and Technology
The Pennsylvania State University
University Park, PA 16802, USA
Email: {lzw158,dwu,pliu}@ist.psu.edu

***Abstract*—Link-based data structures, like linked lists and binary trees, play an important role in organizing and maintaining kernel objects. Attackers have a strong motivation to tamper them for various malicious purposes. Although a lot of techniques have been proposed to protect kernel objects from unauthorized modifications, existing methods can hardly be applied to most kernel data structures. In this paper, we design iCruiser, a novel protection mechanism to universally secure the protected data fields in link-based data structures within kernel. iCruiser introduces a secure canary to guard the protected data fields, and it employs a stream cipher to prevent attackers from compromising the canary field. Without the seed key of the stream cipher, attackers can hardly conduct unauthorized modifications to the protected fields without being detected. Furthermore, to monitor the protection status of iCruiser, we employ the execution trace recording technologies to record the execution of iCruiser, which ensures that any attack attempt happening on iCruiser will be traceable and auditable. Through iCruiser, we can easily narrow down the attacking vectors of link-based kernel data structures for attackers. To show the effectiveness, we applied our design on some critical doubly linked lists in Linux kernel and analyzed the performance overhead at the instruction level.**

I. Introduction

Kernel data structures are important in organizing kernel objects, which provide fundamental building blocks for OS developers in designing kernels. For one hand, kernel objects cooperate together to help kernel achieve its complicate functions; for the other hand, they record the execution states of kernel and provide kernel runtime information for upper-level applications. For example, on a Linux system, users can get a snapshot of current processes by executing command *ps*, which traverses a linked list connecting all the current processes and prints out the process information. If the linked list of the current processes is compromised by an attacker, the process information obtained through *ps* command is not trustworthy. Furthermore, for most of the time, users will not be aware of the compromise.

As a result, attackers would like to compromise kernel data structures to achieve various malicious goals. One goal is to erase attacking trace. An attacker may erase his attacking traces from system logs and hide his existence on the victim machine. Further, he may want to keep his unauthorized privileges by installing a backdoor program [1], [2]. The backdoor program is capable of providing the attacker abilities reinvading the system and retrieving illegal privileges for future attacks. However, the backdoor programs can be easily detected from kernel data structures through system monitor tools. Therefore, attackers have a strong desire to tamper kernel data structures to hide backdoor programs.

Typically, attackers use kernel rootkits [3], [4], [5] to conduct malicious activities in kernel. Kernel rootkit is a kind of malicious program, which modifies kernel stealthily to achieve various malicious goals, including hiding malicious processes, disabling firewalls, changing audit system, and so on. A kernel rootkit can be implemented in many ways. For operating systems supporting loadable kernel module, rootkits are usually implemented with kernel modules, which are running as part of OS kernel. Consider most security tools are user-space applications and will not be able to protect kernel space resources, it is not easy for users to detect malicious activities happened in kernel. A lot of efforts have been done to protect operating systems from being infected by kernel rootkits.

There are mainly three kinds of methods in defending kernel rootkits: detecting kernel object modification, virtual machine inspection, and binary analysis. Detecting kernel object modification [6], [7], [8], [9] makes use of heuristic pre-knowledge of kernel and searches memory space for kernel data and object layout and invariants. Since kernel usually has a fixed memory space layout and many kernel objects will not be changed once established, these ground truths can be used to detect kernel rootkits attacks. Virtual machine inspection [10], [11], [12], [13] takes advantages of virtualization technologies to detect kernel rootkits. Virtual machine monitor (VMM) has a higher privilege than guest OS, which can be used to monitor kernel activities in guest OS. Running at the lowest level, VMM can intercept and retrieve kernel memory pages for security purposes. Binary analysis [14], [15] is another practical method to defend kernel rootkits. Before being loaded into kernel, new kernel modules and device drivers will be profiled with binary analysis to predict potential behaviors. Since malicious kernel modules significantly differ from nonmalicious modules, they can be

detected through binary analysis. Although these methods can help defending kernel rootkits, their protection requires additional supports and resources more or less, which cannot be applied universally.

In this paper, we design iCruiser, a novel protection mechanism to universally secure linked data structures in kernel. Different from existing methods, iCruiser can be universally applied to protect all linked data structures. iCruiser introduces a secure canary to guard the protection fields of a linked node. The protection information is stored in secure canary with stream cipher encryption. The secure canary makes sure that all the unauthorized modifications are detectable. Consider the rootkits enjoy the same privilege with benign code in kernel, there is still a chance for attackers to hack our protection mechanism. To monitor the protection status of iCruiser, we employ an execution trace recording mechanism for auditing purposes. The traces are recorded in read-only mode and can provide sufficient proof for auditing if needed, which ensures attackers cannot escape from being detected.

Our contributions include:

- To our best knowledge, this is the first work reported in open publications that use secure canary to protect other data fields within kernel data structures.
- Among the existing kernel protection methods, iCruiser is the first work to provide a universal way to protect all the link-based data structures in kernel.
- iCruiser is mostly compatible with the legacy kernels and can be used in any mainstream operating systems.
- Our method uses stream cipher to protect kernel data structures, which ensures any unauthorized modification is detectable.
- The security of our protection is traceable. If an attacker is interested in bypassing our protection stealthily, his attempts are recorded and traceable in our execution traces.

We present the threat model in Section II. In Section III, we discuss the motivation of our work. The iCruiser design is presented in Section IV. Section V gives an example of applying our protection on a doubly linked list in Linux kernel and analyzes its possible performance overhead. The security analysis is given in Section VI. We introduce the software cruising methodology employed by iCruiser in Section VII. The related work is given in Section VIII, and conclusion is with Section IX.

II. Threat Model

In this paper, we focus on protecting link-based kernel data structures. Fig. 1 demonstrates our threat model. Our goal is to protect the data field X using secure canary S, a new added data field. As can be seen in Fig. 1, we have a linked-list node A with a protected data field X and a secure canary field S. The secure canary S protects the integrity information of X with a stream cipher. We assume the stream cipher key

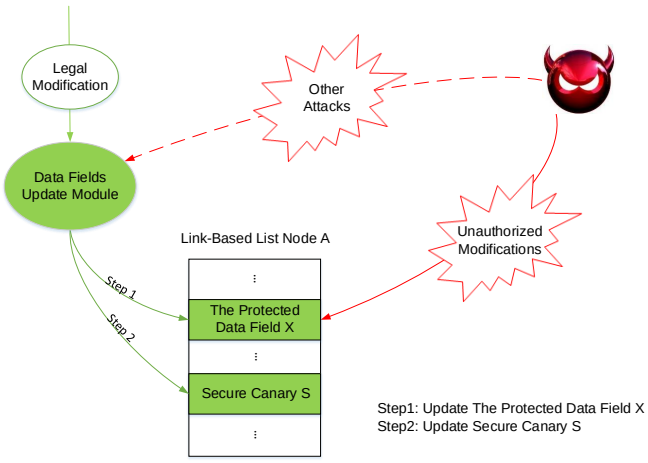

Fig. 1. Threat Model

is well-protected and cannot be obtained by attackers, which can be achieved through certain secure hardware device or VMM. Also, we assume both X and S can be updated legally through a *Data Fields Update Module*. In the *Data Fields Update Module*, the cipher key will be retrieved securely and the secure canary S will be updated based on the latest modification to X. In our assumption, the attacker has root privilege in kernel and wants to tamper the protected field X in A. There are two ways for the attacker to modify the field X in A, *unauthorized modifications* and *other attacks*, which are demonstrated as red arrows in the figure. *Unauthorized modifications* are defined as the attacker directly modifies X by locating A's reference or memory address, which can be achieved in many ways. In unauthorized modifications, since the attacker has no cipher key to update the secure canary, the unexpected modification on X can be detected by checking A's canary field. The *Other Attacks* include returned oriented programming attack [16], [17], code reuse attack [18], [19], and any other attack which is able to bypass iCruiser's protection. The attacker is able to jump to the address of field update module and executes it. Since the modification is made through *Data Field Update Module*, A's canary field is updated legally as well. The modifications achieved by other attacks cannot be detected by checking A's canary field. However, the entire bypassing execution process will be recorded in our execution traces. We leverage previous research results [20] to monitor iCruiser protection, and the entire trace mechanism is trusted and the integrity of the traces is well protected, which ensures all *other attacks* are recorded and traceable. More details will be given in Section IV.

III. Motivation

Open design of modern operating system provides users significant flexibilities in using computer systems. For example, Linux operating system provides users flexible mechanisms to change kernel, like kernel recompiling, loadable kernel module

(LKM), kernel configuration files and tools, device driver, and so on. Taking benefits from these mechanisms, users can dynamically modify running operating systems. Experienced users can even choose to customize a Linux kernel for a special version which can only be used by himself. However, given the flexibilities, an attacker can modify an operating system for malicious purposes as well. The attacker can write a kernel rootkit and load it as a kernel module. In a kernel, the kernel rootkits will be treated with no discriminations and enjoy the same privileges with other legal modules. And, due to kernel rootkits are running at a low level, the mainstream security tools are unable to detect them, which makes regular users can hardly detect attacks brought by kernel rootkits.

Link-based kernel data structures are easy targets of kernel rootkits, for they are widely used in kernel and can be easily obtained and modified by a loadable kernel module. There are a lot of attacking vectors available for attackers to compromise link-based kernel data structures. As we mentioned in Section I, the attacker has a strong desire to tamper kernel data structures. Here are two attacking examples on kernel linked list:

1) **Process Hiding:** Process hiding [21], [9] is a typical attack employed by kernel rootkits to hide a malicious process from system utilities, such as `ps`. It relies on the fact that kernel uses different process lists for scheduling and accounting usages. In Linux, all the processes are linked in a `all-tasks` list, which is headed with `init` process and connects the rest other processes; the process scheduler visits another list `run-list` to schedule processes for execution. To hide a malicious process, attackers can remove it from the `all-tasks` list and keep it in the `run-list` list. As a result, the malicious process can keep running, while the users cannot see it by executing `ps` command.
2) **Tampering Binary Formats:** Inserting a new kernel binary format can help attackers invoke arbitrary malicious code whenever a new process is created. When the kernel generates a new process, a function `search_binary_handler` is called. The function traverses the `formats` list, a global linked list maintaining all binary formats, to locate a proper handler to deal with the new process. It invokes each handler of the `formats` list. If the current handler returns an error code, the function continues with the next handler until a success code is returned. An attacker can insert a new format to the `formats` list and prepare a bad handler carrying malicious code and returning an error code always. Each time the `formats` list is traversed, the bad handler is invoked.

To prevent the *unauthorized modifications* (see Fig. 1), we introduce a secure canary field into link-based kernel data structures. The protected fields are encrypted with a stream cipher and stored in the secure canary field. Each time the protected fields are modified, the secure canary will be recalculated as well. With several simple stream cipher functions, the secure canary could be verified to show whether the protected fields are compromised or not.

IV. Design Overview

To protect link-based kernel data structures in a relatively open kernel environment, several important design choices are made. First, the unauthorized modifications detection should not depend on the reliance from outside of the kernel. We expect to design a self-contained protection method without extra technical supports out of kernel. Second, our protection method should be a typical one which can be universally applied to all link-based data structures. Typically, when we talk about security, secure or not secure is usually described as a feature of the protected object. If we consider security as an attribute of the protected object, we can use a data field of security to store the security information of the protected data fields. Third, we should consider efficiency as well as security. We choose stream cipher encryption to protect the security data field, which is featured with a high efficiency. Last, we should consider the security of our protection mechanism. In case our protection is bypassed or attacked, we considering employing the execution trace recording techniques to secure iCruiser.

A. Secure Canary

Our method inserts an additional data field of security, namely *secure canary*, in the protected data structure. Just as its name implies, *secure canary* guards the protected data fields from being tampered. The *secure canary* should ensure (1) once a compromise happens, the attacker cannot recover the corrupted canary; (2) the canary generation and verification algorithms should be efficient so that they will not hurt kernel performance and detection latency.

We attempt to use simple XOR encryption to compute the secure canary, e.g., a key $\texttt{K} \oplus$ a protected data field `X`. For an attacker, if he tampers `X`, when verifying the secure canary, the calculation of $\texttt{K} \oplus \texttt{X}$ will get a different value instead of the secure canary. Then, we are aware of the compromise. The calculation of canary's value should be protected, which means obtaining the key stream is an extreme difficulty for the attacker. The encryption algorithm and details of calculation will be illustrated in next subsection.

Here, the protected data field could be a link pointer or any protected data field in a link-based data structure. For example, we want to protect the `next_pointer` field of a single link list. The computation of the secure canary should be the corresponding key $\texttt{K} \oplus$ `next_pointer`. We compare the calculated result with the current secure canary field. If they do not match, we know the `next_pointer` field is compromised.

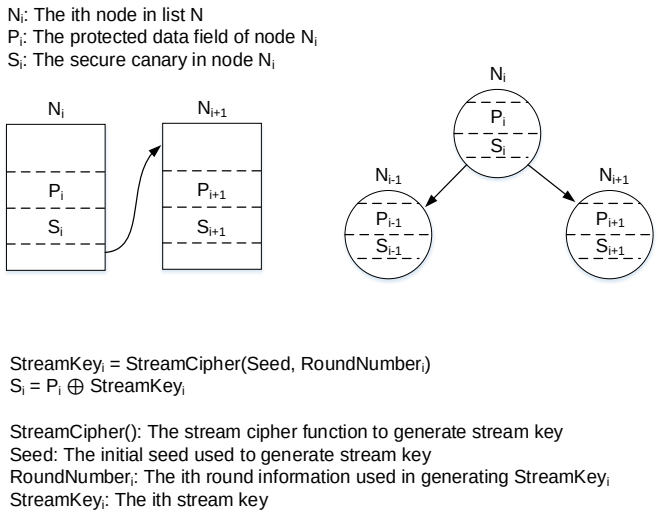

Fig. 2. Protecting Link-Based Data Structures with iCruiser

B. Stream Cipher

We choose stream cipher [22] to calculate the secure canary. Stream cipher is a secure cryptography algorithm, which can be regarded as a highly reliable one-time password [23], [24] system. It uses only two secret information, the seed key and position information, and the seed key could be very short that can be protected without using many resources. It's efficient encryption/decryption operations exactly fit our protection scenarios. As we mentioned in Section II, we assume the seed key is protected well and the attacker cannot steal it. When updating the secure canary, the seed key and the position information will be retrieved securely to calculate the latest canary values. Whenever a verification request is issued, the same calculation will be executed and verified the same way. If an attacker compromised the protected fields, the compromise will be detected when the secure canary is verified.

C. Link-Based Data Structures

When talking link-based data structure, we mean the data structure is constructed through linkages (linkage represents pointer or reference in real programming languages). Fig. 2 shows the examples of protecting link-based data structures with iCruiser. We demonstrate two link-based data structures, link list, and binary tree. As can be seen from the figure, N_i represents the ith node in the list N. In the data fields of node N_i, P_i stands for the protected data field, while S_i represents for the secure canary. The nodes N_i and N_{i+1} are two adjacent nodes linked by a pointer. In N_i, we want to protect P_i with S_i. The symbols used in the figure are designed as follows: $StreamKey_i$ is the ith stream key, $StreamCipher()$ is the stream cipher function to generate key stream, $Seed$ is the initial seed key to generate stream cipher keys, and $RoundNumber_i$ is the round information used in generating $StreamKey_i$.

$$StreamKey_i = StreamCipher(Seed, RoundNumber_i)$$
$$S_i = P_i \oplus StreamKey_i \quad (1)$$

The above equation shows the calculations of the key stream and the secure canary. The $StreamKey_i$ is generated by given two privacies, the $Seed$ and $RoundNumber_i$. The $RoundNumber_i$ information could be easily referred or calculated within node N_i. Our assumption is the $Seed$ is under protection and the attacker cannot get it. Given only $RoundNumber_i$, the attacker cannot manufacture a valid canary field. The secure canary is the encryption result of the $StreamKey_i$ and the protected data field. Stream cipher takes XOR as the encryption and decryption operations. Since the attacker cannot have $Seed$ and further obtain the $StreamKey_i$, he cannot use the right stream key to generate a valid canary field. *We especially note that the protected data field could be any valuable information of the node N_i, such as an important data field or a pointer pointing to next node.*

There is a key assumption in our design, that is we assume the attacker cannot get the $Seed$. Since the kernel rootkits have the same privileges with kernel, it is not safe to store the $Seed$ within kernel. About the details on how to protect the $Seed$, we will give more details in Section V.

V. Protecting Kernel Doubly Linked Lists with iCruiser

A. Kernel Doubly Linked Lists

Doubly Linked List is a representative kernel data structure. It provides both forward and backward circular structures to link data elements. Each DL-List node contains two link fields, which are two pointers pointing to the two adjacent neighbours, the previous node and the next node. Usually, each DL-List is leaded with an initial node, called head node, where the DL-List starts growing. Due to every DL-List node is pointed by its previous and next node, a DL-List can be traversed both in forward and backward directions, which provides a great convenience for a user to visit a certain node.

Doubly Linked List has been widely used in the OS kernel, such as memory organization, process managements, device operations, and file system operations. In memory management, the DL-List is used to organize memory pages; for process management, kernel use different DL-List lists to organize processes.

B. Protecting Kernel DL-Lists with iCruiser

We use iCruiser to protect kernel DL-List. One secure canary will be guarding each protected link field and ensures that all the unauthorized modifications are detectable. The protection is ensured with RC4 [25], a stream cipher using

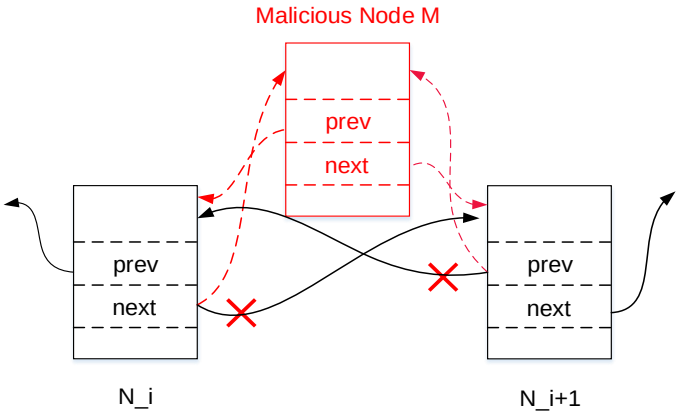

Fig. 3. Inserting a Malicious Node in a DL-List

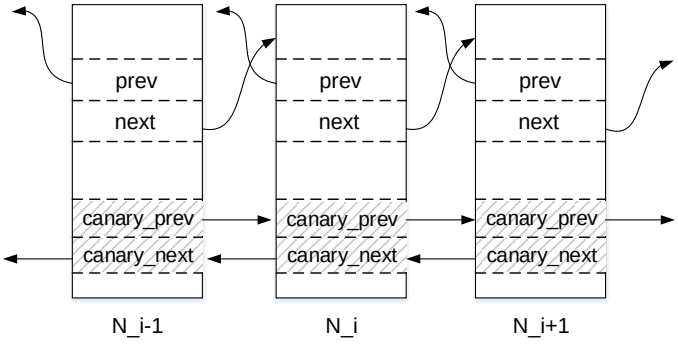

Fig. 4. A DL-List with Linked Secure Canary

symmetric-key cryptograph algorithm. To ensure the protection effectiveness of iCruiser, we employ an existing execution trace recording technique.

1) Unauthorized Modifications: An attacker has a strong desire to compromise the DL-List structure in the kernel so that he can achieve various malicious purposes. Through a loadable kernel module, an attacker can easily obtain a node of `run-list`. Following the forward or backward pointers, he can easily locate the malicious process and remove it from the `run-list`. Consequently, the system utility, like *ps*, will miss the malicious process when displaying the running processes. We call this kind of modification on a DL-List instance as *unauthorized modifications*. Our protection mainly focuses on the unauthorized modification of kernel DL-Lists.

2) Our Protection Goal: The reason why the above attacks succeed is because kernel does not provide a verification mechanism to detect unauthorized modifications on a DL-List. That is, a DL-List node's two pointer fields can only know they "have" two adjacent nodes, but they do not know "who" are their neighbours. We need the DL-Lists nodes are able to verify "who" are their neighbours so that we can detect the unauthorized modifications. Fig. 3 shows inserting a malicious node in a DL-List, and our method should be able to detect the unauthorized modifications like this.

Our goal is to protect and verify the two pointer fields of each DL-List node. With our method, attackers can hardly recover a corrupted DL-List node or generate a new valid DL-List node. Each pointer field will be protected with a secure canary, which securely stores "who" is the authenticate neighbour node. With our protection, an attacker can hardly make unauthorized modifications without being detected.

3) Implementation: We add two secure canary fields in DL-List structure, *canary_prev* and *canary_next*, to protect *prev* and *next* link pointer respectively. Besides, we also add a field *position*, which is used in the RC4 stream cipher to generate a key. The *canary_prev* and *canary_next* fields store the encryption results of *prev* and *next* pointers. Fig. 4 shows a DL-List protected with linked secure canaries. We can see the DL-Lists nodes are also linked by secure canaries. With these two fields, all DL-Lists operations changing *prev* and *next* fields have to recalculate and update secure canary fields. If the attacker still wants to hack *prev* and *next* without being detected, he has to recalculate and update secure canary fields as well, which could be extremely hard under our protection.

To calculate the secure canaries, we use RC4 stream cipher. RC4 stream cipher uses a seed key and S-box to produce a key stream. The seed key is used to initialize the S-box, and the S-box follows an irreversible pseudo-random algorithm to generate a key stream. The pseudo-random algorithm ensures that, as long as the seed key is fixed, the RC4 can always generate a same key stream.

Verifying an unauthorized modification can be easily achieved by checking the related canary fields. For example, we want to verify the node N_i as shown in Fig. 4. First, we can traverse the DL-Lists instance to locate node N_i and retrieve its related fields, *position*, *prev*, and *next*. Then we use the position information to invoke RC4 key stream generation functions to obtain the corresponding keys. With the keys, we can compute the XOR results of its *prev* and *next* field. If the XOR results exactly match N_i's *canary_prev* and *canary_next* fields, no authorized modification happens on N_i. Otherwise, we know an authorized modification happens. Through checking N_i's neighbours, we can further know which node or field is compromised. Similarly, we can verify a whole DL-Lists instance.

C. Performance Analysis

1) Memory Usage: We add three new data fields to the DL-Lists data structure, and each DL-Lists node will need twenty two more bytes (the three data fields are *position, canary_prev,* and *canary_next*) to store a data element. Assume there are one hundred DL-Lists instances in kernel, and each DL-Lists instance may have one hundred nodes. The extra memory used for iCruiser could be hundreds of kilobytes in total. Compared with the memory space controlled by kernel, the amount of extra memory introduced by iCruiser should be acceptable to most users.

2) Computation Overhead: The extra computations introduced by iCruiser are RC4 operations on two new added canary fields. There are two frequently used functions in RC4, key stream generation and encryption (the encryption and decryption are two same operations for RC4 is a symmetric-key cryptograph algorithm). Every time we update a canary, we need to execute these two functions. To verify the canaries, we need to obtain the stream key and run the encryption operation to calculate the latest canary value. If the two values match, we know everything is fine; otherwise, we know there is a compromise.

The heaviest operation in key stream generation function is S-box mutation, which is composed of two basic operations, modulo operation and value switching on two variables. These two basic operations will be translated into several simple binary instructions. The instructions are pretty simple and only takes several instructions cycles. Besides, the encryption/decryption computations of RC4 are just XOR operations, which has been well supported inside of CPU. The new overhead introduced by iCruiser are quite trivial.

D. Execution Trace Recording

Consider attackers may make use of ROP, kernel code reuse, and other attacking methods to bypass the protection of iCruiser. We introduce the execution trace recording mechanism to monitor iCruiser protection status. We want to achieve that, if an attacker successfully hacked iCruiser by using ROP or code reuse attacks, we should be able to audit when it happens and how it finishes. There are many execution logging choices, like deterministic replay [26], [27], OS level logging [28], [29], [30], and specialized hardware [31], [32]. Based on our requirements, we would like to choose XTRec [20], a real-time execution trace recording solutions for commodity systems. XTRec supports instruction-level execution recording and provides evidences to show whether a particular code has been executed, which can be used to the happening of ROP or code reuse attacks.

E. Miscellaneous

We want to talk several practical points when applying iCruiser. The security of the iCruiser is relying on the cryptograph algorithm. In our implementation, we use RC4, and RC4's security guarantee is based on the fact that the attacker can hardly obtain the seed key. In our scenario, we assume the seed key is well protected in TPM and the attacker cannot obtain it anyway.

Also, since the iCruiser is applied in kernel where both malicious program and benign program enjoy the same privilege, we need to deploy some measurements to prevent attackers attacking the iCruiser protections. Here, a successful attacking on iCruiser can be explained like this. If an attacker can succeed compromising the protected fields and iCruiser cannot detect that, we call this a successful attacking. Just as we talked in Section II, there are still other attacks available for an attacker to bypass iCruiser protection. More details will be given in Section VI.

VI. Security Analysis

Our goal is using secure canaries to protect data fields from unauthorized modifications. Through a simple verification, iCruiser is guaranteed to detect whether the unauthorized modifications happened or not. As we analyzed in Section design V, we can achieve our goal with iCruiser. Since we use RC4 to protect secure canaries, the security of iCruiser actually relies on RC4's attack resistance in some degree. The key generation algorithm of RC4 has been proved irreversible and hard to forfeit. As long as the attacker cannot obtain the seed key, it is almost impossible for attackers to hack RC4. We choose to store the seed key in TPM hardware. TPM has been widely deployed on most personal computers and is becoming a standard hardware component. With TPM's protection on the seed key, we can say the stream cipher is well protected and iCruiser's protection is ensured.

However, some attacks are still able to disturb iCruiser's protection, such as Return-Oriented Programming (ROP) [16], [17] and code reuse [18], [19]. An attacker can achieve the modifications by jumping to the start address of the *Data Fields Update Module* and invoke the functions. Or, he can copy the code to a controlled memory region and execute it arbitrarily. Although these attacks have a high attacking bar, well-prepared attackers can still finish it. To monitor the protection status of iCruiser, we execution trace recording technologies. With the execution trace, we can audit whether the *Data Field Update Module* program is executed by an attack or not.

VII. Software Cruising

The iCruiser work is part of a larger project on software cruising [33], [34], [35], which is a novel software monitoring methodology that has been used in heap buffer integrity checking, kernel memory cruising, object invariant checking, data structure protection, rootkit detection, and so on. It makes use of dedicated threads to constantly check and monitor user or kernel programs, for policy enforcement or security violation problems. Following our previous work, Cruiser [33] which detects heap buffer overflows using lock-free algorithms and Kruiser [34] which detects heap buffer overflow in kernel using semi-synchronized algorithms, iCruiser aims to protect link-based data structure in kernel.

Cruiser tries to solve the heap buffer overflow problem by leveraging the popular multiprocessor architectures and lock-free data structures. The basic idea of Cruiser is using a separate monitor thread to detect buffer overflow violations on user threads. The security check of Cruiser is achieved by a parallel thread instead of inline code, which greatly improves the protection performance and reduces the resource reclamations.

Similarly, Kruiser is trying to solve buffer overflow problem within kernel. By using a novel semi-synchronized non-blocking monitoring algorithm, Kruiser is able to detect page-level buffer overflows. Besides, it also uses virtualization technologies to collect the memory information of a guest OS. Kruiser greatly migrates the security enforcement overhead in monitoring kernel buffer overflow problem.

iCruiser follows the software cruising research and tries to protect the data structures. In contrast to Cruiser and Kruiser, iCruiser employs secure canaries to protect the data structures from being compromised by unauthorized modifications. The canary fields can be used to protect any data field or valuable information related with the data structure. In our application, the secure canaries are protected by stream cipher, which prevents the attackers recovering a tampered canary or forging a new valid canary.

VIII. Related Work

Kernel objects modifications usually happen on two kinds of kernel objects, control data and noncontrol data. Control data covers system call tables, system hooks, jump tables, and other objects controlling kernel functions. Noncontrol data includes memory metadata, task list, file inode chains, and so on. SBCFI [6] is proposed to protect kernel from control-flow attacks, which aims to protect the system hooks by periodically checking its memory area and ensure they point to the proper locations. Similarly, Wang et al. [7], [8] propose a lightweight kernel hook protection framework. Based on the observation that the kernel hooks are usually accessed by `read` instead of `write`, they propose to relocate those kernel hooks to a dedicated memory space and protect it with hardware-based protection. While protecting kernel control data is critical, noncontrol data protection is also important. Noncontrol data can help attackers hiding malicious processes, affecting system performance, and disturbing kernel random number generations. To target noncontrol data protection, Petroni et al. [9] present a unified architecture that checks kernel data against a set of integrity specifications, no matter the data is control data or noncontrol data. iCruiser falls in this category as well. Compared with existing methods, iCruiser does not rely on extra support and has an easier implementation and less overhead.

Binary analysis is usually used in analyzing software behaviors. Kruegel et al. [14] first use binary analysis to inspect the kernel modules to be loaded. They collect the behaviors of existing rootkits. If a given module resembles the behavior of a rootkit, this module could be regarded as malicious and will not be loaded into kernel. The method is building on a basis that an abstract model of kernel module behavior will not get affected by small changes in the binary image of the module. Limbo [15] is proposed to protect Windows systems from kernel rootkits. Limbo checks the binary content and run-time behavior of the coming kernel driver and determines whether it is a kernel rootkit. It conducts a characteristic study of the current rootkits and design effective policies for distinguishing benign kernel driver and malicious kernel drivers. While the above binary analysis methods offer helpful options in detecting kernel rootkits, their methods suffer false alarms because the analysis is actually based on conservative approximations.

Virtual machine inspection (VMI) is first provided by Garfinkel et al. [10] in 2003. The initial VMI idea is designed to move the intrusion detection system (IDS) from a host machine to a virtual machine, which helps IDS achieve a better attack resistance. Virtual machine provide an isolation mechanism which can be used for security purposes. Litty et al. [11] designed Manitou, a VMM aided system that helps users arbitrarily modify a program, regardless of the integrity and type of the OS. Manitou makes use of hypervisor privilege to intercept the memory page and modify the permission bits, which can be used to kernel rootkits detections. In addition to the VMI, the virtual machine is also used to monitor the kernel memory space, which can be used to detect kernel rootkits as well. Out-of-VM introspection [12], [13] has been applied to study kernel malware, including kernel rootkits. These protections take the protections through actively monitoring certain kernel memory space. The VM related research provides a lot of valuable methods for protecting kernel from kernel rootkits and other malwares. However, some of them cannot be used to detect kernel data structures. And, these methods require the protected systems are running at virtualization platform, which cannot be applied to many legacy systems.

Besides, some technologies combine two or more techniques to achieve kernel rootkits detections. Gibraltar [36], [14] is an example. It combines kernel object modification and binary analysis to protect kernel. Gibraltar analyzes the entire kernel to locate the vulnerable places and summarize the invariants of the kernel. Here, the invariants mean the properties that are always holding when kernel is running. Through binary analysis, Gibraltar builds up a specification of a health kernel, and compare the observation with the specification. If any observation is found an unexpected kernel object modification, a possible rootkit may happen. Although many rootkits can be detected by Gibraltar, it cannot detect all the compromise on data structures.

IX. Conclusion

In this paper, we have introduced iCruiser, a novel protection mechanism to universally secure all link-based data structures. In iCruiser, we employ a secure canary to guard the protected data fields. The secure canaries are protected with stream cipher. To demonstrate the effectiveness of our method, we have applied iCruiser to protect doubly linked list data structures in Linux kernel. We have analyzed the performance and security effectiveness of iCruiser. Our analysis shows that iCruiser is effective to protect link-based data structures in kernel.

ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation (NSF) Grant CNS-1223710.

REFERENCES

[1] F. Schuster and T. Holz, "Towards reducing the attack surface of software backdoors," in *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS, 2013*, pp. 851–862.

[2] M. F. Abdulla and C. P. Ravikumar, "A self-checking signature scheme for checking backdoor security attacks in internet," *J. High Speed Networks*, vol. 13, no. 4, pp. 309–317, 2004.

[3] A. Bunten, "Unix and linux based rootkits techniques and countermeasures," in *16th Annual FIRST Conference on Computer Security Incident Handling*, 2004.

[4] J. Joy, A. John, and J. Joy, "Rootkit detection mechanism: a survey," in *Advances in Parallel Distributed Computing*. Springer, 2011, pp. 366–374.

[5] J. G. Levine, J. B. Grizzard, and H. L. Owen, "Detecting and categorizing kernel-level rootkits to aid future detection," *IEEE Security & Privacy*, vol. 4, no. 1, pp. 24–32, 2006.

[6] N. L. P. Jr. and M. W. Hicks, "Automated detection of persistent kernel control-flow attacks," in *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS, Alexandria, Virginia, USA, October 28-31, 2007*, pp. 103–115.

[7] Z. Wang, X. Jiang, W. Cui, and P. Ning, "Countering kernel rootkits with lightweight hook protection," in *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS, Chicago, Illinois, USA, November 9-13, 2009*, pp. 545–554.

[8] Z. Wang, X. Jiang, W. Cui, and X. Wang, "Countering persistent kernel rootkits through systematic hook discovery," in *Recent Advances in Intrusion Detection, 11th International Symposium, RAID, Cambridge, MA, USA, September 15-17, 2008. Proceedings*, pp. 21–38.

[9] N. L. P. Jr., T. Fraser, A. Walters, and W. A. Arbaugh, "An architecture for specification-based detection of semantic integrity violations in kernel dynamic data," in *Proceedings of the 15th USENIX Security Symposium, Vancouver, BC, Canada, 2006*.

[10] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proceedings of the Network and Distributed System Security Symposium, NDSS*, 2003.

[11] L. Litty and D. Lie, "Manitou: a layer-below approach to fighting malware," in *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability, ASID, San Jose, California, USA, October 21*, 2006, pp. 6–11.

[12] A. Lanzi, M. I. Sharif, and W. Lee, "K-tracer: A system for extracting kernel malware behavior," in *Proceedings of the Network and Distributed System Security Symposium, NDSS , San Diego, California, USA, 8th February - 11th February 2009*.

[13] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel, "Ensuring operating system kernel integrity with osck," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS , Newport Beach, CA, USA, March 5-11, 2011*, pp. 279–290.

[14] A. Baliga, V. Ganapathy, and L. Iftode, "Detecting kernel-level rootkits using data structure invariants," *IEEE Trans. Dependable Sec. Comput.*, vol. 8, no. 5, pp. 670–684, 2011.

[15] J. Wilhelm and T. Chiueh, "A forced sampled execution approach to kernel rootkit identification," in *Recent Advances in Intrusion Detection, 10th International Symposium, RAID, Gold Goast, Australia, September 5-7, 2007, Proceedings*, 2007, pp. 219–235.

[16] S. Checkoway, L. Davi, A. Dmitrienko, A. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS, 2010*, pp. 559–572.

[17] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: generalizing return-oriented programming to RISC," in *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS, 2008*, pp. 27–38.

[18] T. K. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS, Hong Kong, China, March 22-24, 2011*, pp. 30–40.

[19] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *2013 IEEE Symposium on Security and Privacy, SP, 2013*, pp. 574–588.

[20] A. Vasudevan, N. Qu, and A. Perrig, "XTRec: Secure real-time execution trace recording on commodity platforms," in *44th Hawaii International International Conference on Systems Science (HICSS-44), Proceedings, 4-7 January 2011, Koloa, Kauai, HI, USA*, pp. 1–10.

[21] J. K. Rutkowski, "Advanced windows 2000 rootkits detection," *Blackhat USA*, 2003.

[22] S. Li, X. Mou, and C. Yuanlong, "Pseudo-random bit generator based on couple chaotic systems and its applications in stream-cipher cryptography," in *Proceedings of Progress in Cryptology - INDOCRYPT , Second International Conference on Cryptology in India, 2001*, pp. 316–329.

[23] A. D. Rubin, "Independent one-time passwords," *Computing Systems*, vol. 9, no. 1, pp. 15–27, 1996.

[24] V. Goyal, A. Abraham, S. Sanyal, and S. Han, "The N/R one time password system," in *International Symposium on Information Technology: Coding and Computing (ITCC , Volume 1, 4-6 April 2005, Las Vegas, Nevada, USA*, 2005, pp. 733–738.

[25] P. Ekdahl and T. Johansson, "A new version of the stream cipher SNOW," in *Selected Areas in Cryptography, 9th Annual International Workshop, SAC, St. John's, Newfoundland, Canada, August 15-16, 2002. Revised Papers*, pp. 47–61.

[26] E. O. Voit and J. S. Almeida, "Decoupling dynamical systems for pathway identification from metabolic profiles," *Bioinformatics*, vol. 20, no. 11, pp. 1670–1681, 2004.

[27] F. Baiardi and D. Sgandurra, "Building trustworthy intrusion detection through VM introspection," in *Proceedings of the Third International Symposium on Information Assurance and Security, IAS , August 29-31, 2007, Manchester, United Kingdom*, pp. 209–214.

[28] A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara, "The taser intrusion recovery system," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles , SOSP, 2005*, pp. 163–176.

[29] C. Simache and M. Kaâniche, "Availability assessment of sunos/solaris unix systems based on syslogd and wtmpx log files: A case study," in *11th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC, 12-14 December, 2005, Changsha, Hunan, China*, pp. 49–56.

[30] M. E. Russinovich, D. A. Solomon, and J. Allchin, *Microsoft Windows Internals: Microsoft Windows Server 2003, Windows XP, and Windows 2000*, 4th ed. Microsoft Press, 2005.

[31] S. R. Sarangi, B. Greskamp, and J. Torrellas, "CADRE: cycle-accurate deterministic replay for hardware debugging," in *Proceedings of 2006 International Conference on Dependable Systems and Networks DSN , 25-28 June 2006, Philadelphia, Pennsylvania, USA*, pp. 301–312.

[32] M. T. Yourst, "Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator," in *IEEE International Symposium on Performance Analysis of Systems and Software, April 25-27, 2007, San Jose, California, USA, Proceedings*, 2007, pp. 23–34.

[33] Q. Zeng, D. Wu, and P. Liu, "Cruiser: Concurrent heap buffer overflow monitoring using lock-free data structures," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*, 2011, pp. 367–377.

[34] D. Tian, Q. Zeng, D. Wu, P. Liu, and C. Hu, "Kruiser: Semi-synchronized non-blocking concurrent kernel heap buffer overflow monitoring," in *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS '12)*, 2012.

[35] D. Wu, P. Liu, Q. Zeng, and D. Tian, "Software cruising: A new technology for building concurrent software monitor," in *Secure Cloud Computing*, 2014, pp. 303–324.

[36] A. Baliga, V. Ganapathy, and L. Iftode, "Automatic inference and enforcement of kernel data structure invariants," in *Twenty-Fourth Annual Computer Security Applications Conference, ACSAC , Anaheim, California, USA, 8-12 December 2008*, pp. 77–86.