Dhaka: Protecting Real-Time Embedded Systems Against Data Exploits

Saswata Majumder
Brown University

Abstract

Embedded systems are massively ubiquitous in today's world, with applications in virtually every field due to their flexibility and efficiency. In particular, systems running real-time operating systems (RTOSes) frontline critical tasks that demand tighter temporal guarantees than typical operating systems provide. However, these systems necessarily forego many of the security guarantees seen in general-purpose operating systems, for performance and for the intrinsic difficulty of developing security measures in such a diverse ecosystem of architectures and software. Nevertheless, extensive work has brought defenses against major attack vectors to the RTOS ecosystem. In particular, Kage [5] ensures control-flow integrity [1] for FreeRTOS, an open-source RTOS. However, Kage explicitly does not defend against non-control data attacks.

This work explores the possibility of augmenting Kage with orthogonal memory safety defenses to protect against data attacks, synergising with Kage's memory protection features for much stronger guarantees. We explore hardening schemes at the C source code level, thus easily portable to FreeRTOS, based on the pre-existing memory safety policies SoftBound [11] and CETS [9].

1 Introduction

Over the years, the arms race between software attacks and defenses has generally split into distinct directions. Exploiters find new avenues of attack or reinforce pre-existing ones, to which security researchers respond by developing more specialised defenses, some of which receive the privilege of actual adoption. At the root of such software exploitation lie two major avenues of attack, among others: control-flow hijacking [1] and non-control data attacks [4]. Control-flow hijackers divert the execution of the program in clever ways, while data-only attackers try to silently corrupt data structures without overly disturbing control flow, in both cases realising powerful, even Turing-complete exploitation through limited primitives [13, 6]. In response, the field has developed defenses like control-flow integrity (CFI) to take on the former by restricting program flow to precomputed paths, and memory safety policies for the latter to avoid memory corruption in the first place. However, security policies face a number of other obstacles. For one, they must always balance performance, compatibility, and security in their implementations to lend any practicality - properties that find it difficult to coexist [14]. The defenses themselves are also prone to fundamental weaknesses: CFI, for example, has been shown to be bypassable even in its purest form [13], and data-only attacks prove elusive to schemes with low overhead.

While discussion is often concentrated around general-purpose operating systems, embedded systems also maintain huge portions of the digital world around us with their inherent diversity, simplicity, and efficiency. In particular, some embedded systems run RTOSes for time-critical tasks - for example, Brown Space Engineering is currently working on a space satellite running FreeRTOS[1]. However, as the capabilities of such critical systems increase, such as through greater connectivity through the Internet of Things, so do their attack vectors. This is made worse by the inherent difficulty of protecting embedded systems: many different architectures and software make standardisation of defenses difficult, and their low compute necessitates only defenses with minimal overhead. In fact, such systems forego even the most basic of defenses, such as memory virtualisation in FreeRTOS. Thus, focusing on the development and adoption of defenses for embedded systems is a critical task indeed.

Previous work has focused more generally on embedded systems through mechanisms that ensure return address integrity to prevent backwards edge control-flow hijacking [2]. For example, Silhouette [15] implements efficient shadow stacks for embedded systems using isolation primitives on different architectures. This was later expanded on in Kage [5], a more specialised and comprehensive defense that focuses on ensuring CFI in FreeRTOS while managing overhead. The

[1] https://github.com/BrownSpaceEngineering/PVDXosV2/

name Kage comes from the Japanese word for "shadow", influenced by Silhouette, which likely refers to shadow stacks.

Our work, however, is inspired by the fact that Kage explicitly does not focus on non-control data attacks, but does provide a memory isolation guarantee, which can be utilised by different memory safety schemes. We explore the mechanisms, efficacy, and portability of SoftBound + CETS [10, 8], a combination of defenses for implementing memory safety on general-purpose operating systems using pointer metadata. To this end, we have developed a prototype for instrumentation of C source code with SoftBound + CETS, currently on x86 Linux. Since this prototype uses Kage as a cover for SoftBound and CETS' critical metadata, we call our prototype "Dhaka", which means "covered" in Bangla.

2 Background

2.1 Relevant Attacks and Defenses

Control-flow hijacking [1] attacks target direct or indirect jumps in a program's execution, be it `ret` (backwards edge) or `jmp`/`call` (forwards edge) instructions. They can be combated by ensuring return address integrity using shadow stacks for backwards edge attacks, or pre-computing the control-flow graph (CFG) of the program to restrict forward edge jumps to sets of possible callsites. While effective in drastically reducing the surface area, the callsites left in the legal CFG - necessarily an over-approximation due to the impossibility of perfect disassembly - may still be enough to mount automated attacks [13].

On the other hand, data-only attackers take on the vast surface of data structures in a program, silently corrupting only non-control data such that the program remains within its CFG while still performing computations in favour of the attacker. A simple example is shown below.

```
void auth() {
    int access_level = 0;
    char buf[4];
    const char *pld =
        "\xef\xbe\xad\xde"
        "\xef\xbe\xad\xde"
        "\x01\x00\x00\x00";
    strcpy(buf, pld);
    if (access_level > 0)
        ...
}
```

Here, the buffer overflow corrupts `access_level` to escalate privilege. This causes no violations from a CFI point of view, since the privileged code is still part of the CFG. Thus, data-only attacks are completely orthogonal to control-flow hijacking, so CFI schemes prove ineffective against them.

Numerous defenses against data-only attacks exist, from data randomisation [12] to data-flow integrity (DFI) [3], although some incur large overhead due to the complexity of comprehensively protecting data. One such class of defenses is memory safety, which ensures memory cannot be corrupted in the first place. Spatial memory safety ensures memory accesses do not go out of allocated bounds to prevent information leakage or corruption, and temporal memory safety ensures supposedly free regions of memory cannot be accessed in the future through loose ends.

Memory safety can in turn be implemented in a number of ways, from memory-safe languages from the ground-up [7] to defenses that add instrumentation to validate memory accesses. One such combination of defenses is SoftBound and CETS: the former ensures spatial safety, the latter temporal. SoftBound stores lowest and highest address values as metadata for every allocated pointer, and performs checks on pointer accesses to ensure they do not stray out of bounds. Meanwhile, CETS associates a unique key and lock address pair with every pointer to a different location, with the lock being initialised with the key - a pointer access is only valid if its key still matches its lock's value. Since a pointer's lock is set to be invalid when the pointer is deallocated, this ensures loose references fail the key-lock check. Together, these ensure full memory safety at the cost of high overhead due to the instrumentation - also assuming the data structures used to store the pointer metadata are somehow secured.

2.2 Real-Time Embedded Systems

Embedded systems are minimal, efficient, specialised digital systems meant for specific purposes, for example a drone. However, some tasks require greater timing accuracy than operating systems generally provide for embedded and general-purpose systems alike, such as a space satellite that needs to constantly receive signals while also validating its own systems in the background. This is where FreeRTOS comes in. At its core, it is an open-source RTOS that supports different classes of microcontrollers using the same API, running efficiently on bare metal but also with the ability to enable different security features. More importantly, the FreeRTOS kernel schedules different processes, called "tasks", to run at different times on the same processor, like general-purpose operating systems. Unlike those, however, FreeRTOS also encodes priorities for each task, such that at any given time only the task with the highest priority gets to run, thus ensuring important tasks are able to respond more reliably than more background processes.

Due to their simplicity, embedded systems generally lack the defense mechanisms we take for granted on general-purpose systems, such as memory virtualisation or isolation. FreeRTOS does provide the optional Memory Protection Unit to implement privilege levels and isolation, although it introduces noticeable overhead and does not provide full security

guarantees. In light of this, different defenses have evolved specifically for bare metal applications, such as the aforementioned return address integrity (RAI), Silhouette, and especially Kage. Kage focuses on FreeRTOS to provide CFI in a number of ways, building on Silhouette's efficient shadow stacks and its own efficient memory isolation mechanisms using the MPU to enforce a number of guarantees needed to ensure CFI. While CFI policies are known to be imperfect even in an ideal setting, due to the small scale of code available to FreeRTOS programs, evaluations have shown control-flow hijacks to be not possible in tested cases, as Kage reduces enough of the surface area to make an attack unfeasible.

Our work aims to expand on this using SoftBound + CETS to protect against the other looming vector of data-only attacks. Of course, the specific scheme used to protect against data-only attacks can be changed for performance or other reasons - we aim to demonstrate that combining these defenses would indeed lead to more comprehensive coverage. What makes this combination particularly enticing is that SoftBound + CETS rely on memory isolation to avoid the pointer metadata from being leaked or corrupted, and Kage handily provides a mechanism for us to isolate memory from unprivileged, possibly compromised tasks. Thus, Kage would ensure the efficacy of SoftBound + CETS in implementing memory safety by protecting the only vulnerable data, while SoftBound + CETS cover Kage's blind spot in terms of data-only attacks.

3 Design

3.1 Threat Model

We consider an embedded system running Kage-FreeRTOS with the MPU enabled, meaning CFI attacks are very unlikely to be feasible or expressive. Our adversary has control of a compromised unprivileged task on the system, and has access to at least one memory corruption vulnerability that can be triggered repeatedly, except for accessing the privileged memory protected by Kage.

3.2 Implementation

Kage is currently only implemented for a specific microcontroller board, and FreeRTOS itself can only be emulated on x86 Linux. Thus, we aim to harden C source code using SoftBound + CETS, compiled on x86 32-bit Ubuntu without address space randomisation and other defenses to best mimic a typical FreeRTOS system. This is because FreeRTOS is implemented entirely in C - including this prototype into a FreeRTOS project is a matter of figuring out Makefiles.

Combining SoftBound and CETS in our Dhaka prototype, lovingly aliased as Shyamoli and Chhayanaut respectively, we require one data structure to map pointed addresses to their metadata for two types of checks. Ideally, we want to associate these pieces of information with every pointed address:

```
typedef struct {
    uint64_t key;
    uint64_t* lock_addr;
    uintptr_t base_addr;
    uintptr_t bound_addr;
    PtrFreeable freeable;
} ptr_meta;
```

`key` and `lock_addr` are associated with CETS, while `base_addr` and `bound_addr` allow for SoftBound checks. Meanwhile, the pseudo-boolean `freeable` should be used to track which pointers need to be freed, and which are simply references not returned by a call to `malloc` or similar.

Dhaka would thus instrument the source code in 3 different situations, all detailed in the GitHub repository [2]:

- **Pointer creation:** either after a call to `malloc` or pointer arithmetic, Dhaka should add the pointer's metadata to the data structure. The metadata will differ based on the nature of the pointer. A `malloc`'d pointer would simply be assigned a new key-lock pair, have its own pointed address be the base address, and its base + pointer access size (the size of the pointed datatype) as the upper bound, while being marked as `FREEABLE`. Meanwhile, pointers borne of arithmetic (including simple pointer duplication) will inherit most information from the pointer it is based on except for being marked as unfreeable, since legal pointer arithmetic mostly involves traversing an allocated data structure. No checks are performed at this stage, even if the operation is undefined behaviour.

- **Pointer access:** when a pointed address is accessed for any reason, Dhaka will insert itself to perform an access check. It will look up the pointer's metadata in the data structure, and separately perform the softBound and CETS checks. This entails a bounds check for SoftBound, and a comparison between the pointer's lock and key for CETS; if any check fails, a customisable error handler is immediately called.

- **Pointer deallocation:** when freeing a pointer, its lock should be set to `INVALID_KEY`, which will not match with any key as the key increments starting after `INVALID_KEY`. At the end, the metadata entries should be deallocated and popped from the data structure, and any freeable pointers should be freed exactly once. This will be done by manually finding all tracked pointers and deleting them at the end.

Our implementation consists of C source code implementing the metadata structure and the macros to wrap

[2] https://github.com/Saswater/Dhaka-FreeRTOS

pointer operations with, comprising $\sim$ 250 lines. Additionally, `posix_text.c` contains normal vs. hardened examples for different situations, including commented-out errors to demonstrate either Shyamoli (SoftBound) or Chhayanaut (CETS) working. With the power of macros, we have reduced instrumentation to one line inserted before (for pointer access) or after (for pointer creation) the pointer operation, from the set `dhalloc` (Dhaka alloc), `LMAP_ADD`, `CHK_ACCESS`, and `LMAP_DEL`.

We have chosen to implement the metadata structure as a global linked list with each node being a key-value pair, where the key is the address pointed to by a pointer, and the value is `ptr_meta` above. This differs from the original paper's implementation that simply added the metadata as local variables around the pointer, to make situations like passing pointers through functions easier - the paper had to ensure the local variables would be passed along as well. With our implementation, every pointer can be queried as required. Additionally, this linked map (`lmap`) data structure stores new entries and their associated data on the heap for simplicity in this demo. When combining Dhaka with Kage or any other memory-isolation defense like FreeRTOS's MPU, storing this data structure and all of its dynamic allocations in the privileged memory section should help in ensuring memory safety. Lastly, lookup times in the linked map are linear with respect to the number of tracked pointers, as we have chosen a proof of concept over speed. The data structure can be easily swapped out for a hashmap or similar structure to ensure faster pointer lookups, as the instrumentation is sure to introduce overhead.

As an example, if we try to use pointer arithmetic to access outside of a pointer's associated area, we get:

```
Spatial memory violation detected by Shyamoli:
0xbffffe1d is out of the bounds
0xbffffe16 to 0xbffffe1c!
Line: 31, File: dhaka.c
make: *** [Makefile:36: 1run] Error 255
```

And if we try to access a dangling pointer, we get:

```
Temporal memory violation detected by Chhayanaut:
0x804c600's key (3) and lock (14521449586448433228)
don't match!
Line: 30, File: dhaka.c
make: *** [Makefile:36: 1run] Error 255
```

4 Preliminary Evaluation and Further Work

Dhaka is effective at detecting spatial and temporal memory safety violations with minimal instrumentation at the C source code level, requiring only single-line changes for many situations. It is thus able to protect against data-only attacks without interfering with CFI-based defenses such as Kage, and will in fact benefit from Kage's efficient memory isolation schemes. However, not all pointer interactions are supported or have been tested. Using pointer arithmetic to access other members of a struct is not allowed, among other potential style changes. Objects like strings are also much harder to take into consideration, considering their null-terminated nature that makes virtually every string operation unsafe.

Including our prototype in the x86 Linux simulator for FreeRTOS is our next immediate priority, to demonstrate how it would slot naturally into a FreeRTOS program due to not requiring any specific coding standards or libraries. With that done, we can look into expanding support for different pointer situations. If possible, we will borrow a compatible microcontroller to load Dhaka-hardened FreeRTOS programs into.

Due to the simplicity of the hardening at times, a script could potentially perform the hardening automatically. We may initially look into regex and similar parsers for C programs, and then turn to more abstract representations such as Clang's Abstract Syntax Tree (AST) for easier tooling and the possibility of an end-to-end tool.

5 Conclusion

We explored Dhaka, a prototype for implementing memory safety at the source code level in Kage, a CFI-hardened version of FreeRTOS for embedded systems. Our prototype uses SoftBound + CETS for the memory safety scheme, with its critical metadata structures being protected by Kage's memory isolation capabilities. This proposal highlights the importance of exploring and widening the coverage of embedded systems security in today's world.

Availability

Our prototype of Dhaka is available at:
https://github.com/Saswater/Dhaka-FreeRTOS

Acknowledgements

I wholeheartedly thank Professor Vasilis Kemerlis for his unwavering guidance throughout an entire academic year of software security, for organically introducing me into a field I would not have known I have an interest in, and for his diligence and skill in creating what I deem to be one of my most educational semesters, if not the one. His assistance with Dhaka in response to my personal circumstances will also not go unappreciated.

References

[1] Martín Abadi et al. "Control-flow integrity". In: *Proceedings of the 12th ACM Conference on Computer and Communications Security*. CCS '05. Alexandria, VA, USA: Association for Computing Machinery, 2005, pp. 340–353. ISBN: 1595932267. DOI: 10.1145/1102120.1102165. URL: https://doi.org/10.1145/1102120.1102165.

[2] Naif Almakhdhub et al. "μRAI: Securing Embedded Systems with Return Address Integrity". In: Jan. 2020. DOI: 10.14722/ndss.2020.24016.

[3] Miguel Castro, Manuel Costa, and Tim Harris. "Securing software by enforcing data-flow integrity". In: *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*. OSDI '06. Seattle, WA: USENIX Association, 2006, p. 11.

[4] Shuo Chen et al. "Non-control-data attacks are realistic threats". In: *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*. SSYM'05. Baltimore, MD: USENIX Association, 2005, p. 12.

[5] Yufei Du et al. "Holistic Control-Flow Protection on Real-Time Embedded Systems with Kage". In: *Proceedings of the 31st USENIX Security Symposium*. Security '22. Boston, MA, USA: USENIX Association, 2022, pp. 2281–2298. ISBN: 978-1-939133-31-1. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/du.

[6] Kyriakos K. Ispoglou et al. "Block Oriented Programming: Automating Data-Only Attacks". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. Toronto, Canada: Association for Computing Machinery, 2018, pp. 1868–1882. ISBN: 9781450356930. DOI: 10.1145/3243734.3243739. URL: https://doi.org/10.1145/3243734.3243739.

[7] Nicholas D. Matsakis and Felix S. Klock. "The rust language". In: *Ada Lett.* 34.3 (Oct. 2014), pp. 103–104. ISSN: 1094-3641. DOI: 10.1145/2692956.2663188. URL: https://doi.org/10.1145/2692956.2663188.

[8] Santosh Nagarakatte et al. "CETS: compiler enforced temporal safety for C". In: *Proceedings of the 2010 International Symposium on Memory Management*. ISMM '10. Toronto, Ontario, Canada: Association for Computing Machinery, 2010, pp. 31–40. ISBN: 9781450300544. DOI: 10.1145/1806651.1806657. URL: https://doi.org/10.1145/1806651.1806657.

[9] Santosh Nagarakatte et al. "CETS: compiler enforced temporal safety for C". In: *SIGPLAN Not.* 45.8 (June 2010), pp. 31–40. ISSN: 0362-1340. DOI: 10.1145/1837855.1806657. URL: https://doi.org/10.1145/1837855.1806657.

[10] Santosh Nagarakatte et al. "SoftBound: highly compatible and complete spatial memory safety for c". In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '09. Dublin, Ireland: Association for Computing Machinery, 2009, pp. 245–258. ISBN: 9781605583921. DOI: 10.1145/1542476.1542504. URL: https://doi.org/10.1145/1542476.1542504.

[11] Santosh Nagarakatte et al. "SoftBound: highly compatible and complete spatial memory safety for c". In: *SIGPLAN Not.* 44.6 (June 2009), pp. 245–258. ISSN: 0362-1340. DOI: 10.1145/1543135.1542504. URL: https://doi.org/10.1145/1543135.1542504.

[12] Prabhu Rajasekaran et al. "CoDaRR: Continuous Data Space Randomization against Data-Only Attacks". In: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. ASIA CCS '20. Taipei, Taiwan: Association for Computing Machinery, 2020, pp. 494–505. ISBN: 9781450367509. DOI: 10.1145/3320269.3384757. URL: https://doi.org/10.1145/3320269.3384757.

[13] Victor van der Veen et al. "The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 1675–1689. ISBN: 9781450349468. DOI: 10.1145/3133956.3134026. URL: https://doi.org/10.1145/3133956.3134026.

[14] Xiaoyang Xu et al. "CONFIRM: Evaluating Compatibility and Relevance of Control-flow Integrity Protections for Modern Software". In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1805–1821. ISBN: 978-1-939133-06-9. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/xu-xiaoyang.

[15] Jie Zhou et al. "Silhouette: efficient protected shadow stacks for embedded systems". In: *Proceedings of the 29th USENIX Conference on Security Symposium*. SEC'20. USA: USENIX Association, 2020. ISBN: 978-1-939133-17-5.