BUDAlloc: Defeating Use-After-Free Bugs by Decoupling Virtual Address Management from Kernel

Junho Ahn, Jaehyeon Lee, Kanghyuk Lee, Wooseok Gwak, Minseong Hwang, and Youngjin Kwon, *KAIST*

https://www.usenix.org/conference/usenixsecurity24/presentation/ahn

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 33rd USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

USENIX Security '24 Artifact Appendix
BUDAlloc: Defeating Use-After-Free Bugs by Decoupling Virtual Address

Junho Ahn, Jaehyeon Lee, Kanghyuk Lee, Wooseok Gwak, Minseong Hwang, Youngjin Kwon
School of Computing, KAIST

A Artifact Appendix

This paper introduces BUDAlloc, a one-time-allocator for detecting and protecting use-after-free bugs in unmodified binaries. The core idea is co-designing a user-level allocator and kernel by separating virtual and physical address management. The user-level allocator manages virtual address layout, eliminating the need for system calls when creating virtual aliases. This is essential for reducing internal fragmentation caused by the one-time-allocator. BUDAlloc customizes the kernel page fault handler with eBPF for batching unmap requests when freeing objects. In SPEC CPU 2017, BUDAlloc achieves a 15% performance improvement over DangZero and reduces memory overhead by 61% compared to FFmalloc.

A.1 Abstract

This artifact evaluation provides the source code, runtime setup, and instructions needed to reproduce the BUDAlloc evaluation results. We evaluate BUDAlloc in terms of security, performance, and memory usage. For the security evaluation, we conduct CVE analysis, use the NIST Juliet test suite, and HardsHeap. For performance evaluation, we test BUDAlloc with SPEC CPU 2006 & 2017, PASEC 3.0, Apache, and Nginx Webserver. This artifact demonstrates that BUDAlloc effectively prevents and detects use-after-free (UAF) bugs while having minimal impact on performance and memory usage.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

There are no ethical concerns associated with BUDAlloc. The source code is released under the MIT license.

A.2.2 How to access

This artifact evaluation can be accessed via the following stable URL: https://github.com/casys-kaist/BUDAlloc/tree/9adddb369c2e74e86431459c627417f2f57cadbb.

A.2.3 Hardware dependencies

We tested BUDAlloc with Intel(R) Xeon(R) Gold 5220R CPU at 2.2GHz with 24 cores, 172GB DRAM - 2666 MHZ, 512 GB SSD, and 10-Gigabit Network Connection. In all the experiments, we disable hyper-threading, CPU power-saving states, and frequency scaling to reduce the variance. We use Non-Uniform Memory Access (NUMA) in the PARSEC 3.0 benchmarks to fully utilize all 48 cores in the motherboard. We use time to get the resident set size (RSS) and total execution time except DangZero.

A.2.4 Software dependencies

To support atomic operations in the BPF program, BUDAlloc requires the installation of clang-17. This can be done using the 'scripts/setup.sh' script. Our setup utilizes Ubuntu 20.04 with GCC version 9.4.0. If using a newer version of GCC, the `-fcommon` and `-Wno-implicit-function-declaration` compiler options are necessary. We use default configurations for other memory allocators in all evaluations. For testing on DangZero, we use a virtual machine with KVM, as this is the default method for running Kernel-Mode-Linux in DangZero.

A.2.5 Benchmarks

We used SPECCPU 2006, SPECCPU 2017, PARSEC 3.0, Apache2, Nginx to benchmark the performance of BUDAlloc. To evaluate the robustness of BUDAlloc compared to other OTA systems, we evaluate a set of Common Vulnerabilities and Exposures (CVEs), HardsHeap Fuzzer, and NIST Juliet Test Suite.

A.3 Set-up

BUDAlloc consists of two distinct components: the kernel and the user space. The BUDAlloc kernel includes the necessary kernel patches for the eBPF helper functions and custom page fault handler. The BUDAlloc user space contains both the user-level components and the eBPF custom page fault handler.

A.3.1 Installation

You can find more information in our github repository.

BUDAlloc-Kernel Installation.

1. Clone the BUDAlloc repository

```
git clone
 ↪ https://github.com/casys-kaist/BUDAlloc
```

2. Get submodules and update. This will clone BUDAlloc-Kernel repository.

```
$ cd BUDAlloc
$ git submodule init
$ git submodule update
```

3. Build and install the kernel. In the kernel configurations, `CONFIG_BPF_SBPF` should be enabled, and `CONFIG_BPF_SBPF_MEM_DEBUG` should be turned off to measure accurate performance.

```
$ cd BUDAlloc-Kernel
$ make -j$(nproc)
$ sudo make -j$(nproc) INSTALL_MOD_STRIP=1
 ↪ modules_install
$ sudo make install
```

4. Reboot your system.

5. After rebooting, install the `libbpf` library.

   (a) Navigate to the `libbpf` directory and build the library.

   ```
   $ cd BUDAlloc-Kernel/tools/lib/bpf
   $ make -j$(nproc)
   $ sudo make install
   ```

   (b) Install the kernel header files.

   ```
   $ cd BUDAlloc-Kernel
   $ sudo make headers_install
    ↪ INSTALL_HDR_PATH=/usr
   ```

6. Enable linking for `libbpf`.

```
$ sudo vi /etc/ld.so.conf.d/99.conf
# add "/usr/local/lib64"
$ sudo ldconfig
```

BUDAlloc-User Installation. You should proceed this part after installing `BUDAlloc-Kernel`.

1. Install the Clang-17 compiler.

```
$ ./scripts/setup.sh
```

2. Build and install the user components.

```
$ make -j$(nproc)
$ sudo make install
```

[Note]: Default build is `BUDAlloc-p(revent)` mode. To build `BUDAlloc-d(etect) mode`, follow below instruction.

```
$ vim libkernel/include/kconfig.h
# comment out:
# #define CONFIG_BATCHED_FREE
# #define CONFIG_ADOPTIVE_BATCHED_FREE
$ make -j$(nproc)
$ sudo make install
```

Installation guide for the related works. We used `ffmalloc`, `MarkUS` and `Dangzero` libraries for our evaluation. You can get an installation guide for each library in the links below.

1. ffmalloc

   `https://github.com/bwickman97/ffmalloc`

2. MarkUS

   `https://github.com/MarkUsProject/Markus`

3. Dangzero

   `https://github.com/vusec/dangzero`

A.3.2 Basic Test

After installing BUDAlloc-Kernel and BUDAlloc-User, you can test your program with the following script in the BUDAlloc repository.

```
make unit_test
```

A.4 Evaluation workflow

We evaluated the **performance, memory overhead, and bug detectability** of BUDAlloc compared to recent OTAs, FFmalloc, DangZero, and the GC-based MarkUs. In SPEC CPU 2006, BUDAlloc outperformed DangZero by 5% in full detection mode and by 15% in prevention mode, with a memory overhead of 30% compared to FFmalloc's 207%, and better bug detectability. BUDAlloc showed scalable performance improvements in multithreaded PARSEC 3.0, surpassing FFmalloc with more than 8 threads. Real-world tests with Nginx and Apache demonstrated performance and memory overhead comparable to GLIBC, without scalability issues. BUDAlloc detected 27 out of 30 use-after-free vulnerabilities from recent CVEs, and passed all robustness tests with Fuzzer and NIST Juliet, with no issues found in HardsHeap after 24 hours.

A.4.1 Major Claims

(C1): BUDAlloc should demonstrate acceptable performance and memory overhead on single-thread benchmarks such as SPEC CPU 2006 and SPEC CPU 2017.
(C2): BUDAlloc should show scalable performance improvements on multi-thread benchmarks such as PARSEC 3.0.
(C3): In prevention mode, BUDAlloc should successfully prevent all use-after-free and double-free bugs in the Juliet, HardsHeap, and CVE corpus.
(C4): In detection mode, BUDAlloc should detect all use-after-free and double-free bugs in the CVE sets.

A.4.2 Experiments

All results will be stored in `macrobench/result/<bench>`. Before running the script, we recommend extending the sudo authentication timeout.

```
$ sudo visudo
# Add the line "Defaults:<User_name>
↪ timestamp_timeout=600"
$ sudo -k
```

Bench script options In each script, you may configure options according to your preferences. The available options are as follows:

1. `--LIBCS`: Set library(s) to run. Default value is `"glibc BUDAlloc ffmalloc markus"`, which will run 4 libraries sequently.

2. `--TASKSET`: Set thread number of `taskset` command. Default value is `19`. which is required to bind the core and reduce fluctuation. This also limits the additional CPU resources consumed by MarkUs's GC thread, unlike other test cases.

3. `--THREADS`: **[Only for PARSEC 3.0]** Set the number of threads to run. The default value is `"1 2 4 8 16 32"`, which will run program with 1, 2, 4, 8, 16, and 32 threads, sequentially.

4. `--CONNECTIONS`: **[Only for Apache2 and Nginx]** Set connection number of benchmark. Default value is `"100 200 400 800"`, which will run 100, 200, 400, 800 connections sequently.

5. `--BENCH_SEC`: **[Only for Apache2 and Nginx]** Set the connection time for benchmark. Default value is `30`.

Additionally, if you are testing `DangZero`, you should set `--LIBCS=dangzero` in each benchmark script.

```
$ ./bench_xxx.sh --LIBCS=dangzero
```

Memory usage of DangZero. DangZero cannot account for the memory usage with the default scripts. Unlike other benchmarks, to get a resident set size, you have to add the additional value by uncommenting the `TRACK_MEM_USAGE` option.

(E1): *[SPECCPU 2006] [1 human-minutes + 1.5 compute-hour/lib + 3.5GB disk]: This will run SPECCPU_2006 for each library. Full test will test 4 libraries(glibc, BUDAlloc, ffmalloc, markus)*
Preparation: Before starting, you should obtain and install `SPECCPU 2006` in the `/home/<USER>` directory.
Execution: Execute `bench_spec2006.sh` to run SPECCPU 2006 benchmarks.

```
$ cd macrobench/spec2006
$ ./bench_spec2006.sh [OPTIONS]
```

Results: Results will be located in `result_<library_name>_<INT/FlOAT>.csv`

(E2): *[SPECCPU 2017] [1 human-minutes + 7 compute-hour/lib + 11GB disk]: This will run SPECCPU 2017 for each library. Full test will test 4 libraries(glibc, BUDAlloc, ffmalloc, markus).*
Preparation: Before starting, you should obtain and install `SPECCPU 2017` in the `/home/<USER>` directory.
Execution: Execute `bench_spec2017.sh` to run `SPECCPU 2017` benchmarks.

```
$ cd macrobench/spec2017
$ ./bench_spec2017.sh [OPTIONS]
```

Results: Results will be located in `result_<library_name>_<INT/FlOAT>.csv`

(E3): *[PARSEC 3.0] [1 human-minutes + 1-3 compute-hour per cases, depending on library and thread numbers + 13GB disk]: This will run PARSEC 3.0 benchmark for each library, also per different thread numbers. Full test will test 4 libraries(glibc, BUDAlloc, ffmalloc, markus) with 6 different thread numbers(1, 2, 4, 8, 16, 32).*
Preparation: Install `PARSEC 3.0` benchmark.

```
$ cd macrobench/parsec
$ sudo ./install.sh
```

Execution: Execute `bench_parsec_threads.sh` in `macrobench/parsec` to run `PARSEC 3.0` benchmarks.

```
$ cd macrobench/parsec
$ sudo ./bench_parsec_threads.sh
```

Results: Results will be located in the following file: `result_<library_name>_<thread_num>t.csv`

Note for (E4), (E5): For **(E4)** and **(E5)**, you should first set up a 10G Ethernet connection between the server machine and client machine. Make sure to adjust the variables in `macrobench/server_conf` according to your environment.

```
# Host
```

```
$ cd macrobench/common
$ ./connect_10g server

# Client
$ cd macrobench/common
$ ./connect_10g.sh client
```

(E4): *[Apache2] [1 human-minutes + 0.5 compute-hour + 4GB disk]: This will run Apache2 benchmarks for each library.*
Preparation: Install `Apache2` webserver.

```
$ cd macrobench/apache2
$ ./install.sh
```

Execution: Execute `Apache2` benchmark.

```
$ cd macrobench/apache2
$ sudo ./bench_apache2.sh
```

If scripts don't run as intended, you can manually run host and client on each server. By specifying `"foreground"` for the last argument of `run_apache_nginx_server.sh`, it will run `Apache2` server in the foreground.

```
# Host
$ cd macrobench/common
$ sudo ./run_apache_nginx_server.sh
 ↪ <apache2/nginx> <library> <num_threads>
 ↪ foreground
# Client
$ cd macrobench/common
$ sudo ./run_apache_nginx_client.sh
 ↪ <apache2/nginx> <Num_connections>
 ↪ <num_threads> <BENCH_SEC> <library>
```

Results: The results will be divided into two parts. Performance (latency) will be recorded in `result_xxx.csv`, and memory usage will be recorded in `apache2_memory_usage_xxx.csv`.

(E5): *[Nginx] [1 human-minutes + 3 compute-minute + 4GB disk]: This will run Nginx benchmarks for each library.*
Preparation: Install `Nginx` webserver.

```
$ cd macrobench/nginx
$ ./install.sh
```

Execution: Execute `Nginx` benchmark.

```
$ cd macrobench/nginx
$ sudo ./bench_nginx.sh
```

Results: The results will be separated into two parts. Performance (latency) will be recorded in `result_xxx.csv`, and memory usage will be recorded in `nginx_memory_usage_xxx.csv`.

(E6): *[CVE] [1 human-minutes + 30 compute-minute]: This will run CVEs and check detect/prevent/vulnerable features for each library.*

Note: Our script cannot automatically classify all CVEs because some UAF bugs occur internally or do not produce any output. In these cases, we classified them as `CANNOT_DETERMINE`. For more precise checking, please compile each program with AddressSanitizer or use another method to detect Use-After-Free bugs. Detailed information is provided in the `README` of each CVE.
Also, you should care about `prevent` and `detect` mode in BUDAlloc. If you want to change the mode, please refer to the **[Note]** in **BUDAlloc installation** section.
Preparation: Before starting, you should install all CVE-related programs. You can easily install them with the following command:

```
$ cd validation/cves/install_lib.sh
$ make
# If make stops,
# Please use this command
# This will build programs sequently
$ make build_serial
```

Execution: Run all CVEs and get the result.

```
$ make run
```

Results: The results will be located in the following file: `validation/cves/result.csv`.

(E7): *[Juliet Suite]* [1 human-minutes + 5 compute-minute]: This will run Juliet Suite and check the correctness of the allocator.
Preparation: Install `Juliet Suite` benchmark.

```
$ validation/juliet/install.sh
```

Execution: Execute `Juliet Suite`.

```
$ validation/juliet/bench_juliet.sh
```

Results: Executing `Calling good` should be passed, as they are valid programs, and `Calling bad` should incur segmentation fault, and they are invalid programs.

(E8): *[HardsHeap]* [1 human-minutes + 24 compute-hour]: This will run HardsHeap Fuzzer and check the correctness of the allocator.
Execution: Execute `HardsHeap` benchmark.

```
$ validation/hardsheap/run_hardsheap.sh
```

Results: After hardsheap execution, there should be no invalid test cases.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.