УНИВЕРСИТЕТ ИТМО
НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
январь–февраль 2019 Том 20 № 1 ISSN 2226-1494 http://ntv.itmo.ru/
SCIENTIFIC AND TECHNICAL JOURNAL OF INFORMATION TECHNOLOGIES, MECHANICS AND OPTCS
January–February 2020 Vol. 20 No 1 ISSN 2226-1494 http://ntv.itmo.ru/en/
НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК УНИВЕРСИТЕТ ИТМО
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

УДК 004.052.42 doi: 10.17586/2226-1494-2020-20-1-101-109

ПОДХОД К ВЕРИФИКАЦИИ АЛЛОКАТОРОВ ДИНАМИЧЕСКОЙ ПАМЯТИ, ОСНОВАННЫЙ НА СИМВОЛЬНОМ ВЫПОЛНЕНИИ ПРОГРАММ

А.М. Дергачев, Д.С. Садырин, А.Г. Ильина, И.П. Логинов, Ю.Д. Кореньков

Университет ИТМО, Санкт-Петербург, 197101, Российская Федерация
Адрес для переписки: dam600@gmail.com

Информация о статье
Поступила в редакцию 03.12.19, принята к печати 30.12.19
Язык статьи — русский

Ссылка для цитирования: Дергачев А.М., Садырин Д.С., Ильина А.Г., Логинов И.П., Кореньков Ю.Д. Подход к верификации аллокаторов динамической памяти, основанный на символьном выполнении программ // Научно-технический вестник информационных технологий, механики и оптики. 2020. Т. 1. № 1. С. 101–109. doi: 10.17586/2226-1494-2020-20-1-101-109

Аннотация

Предмет исследования. Исследованы техники эксплуатации уязвимостей в реализации алгоритмов распределения динамической памяти – аллокаторе библиотеки glibc (Poisoned Null-byte, Overlapped Chunks, Fastbin Attack, Unsafe Unlink, House of Einherjar, House of Force, House of Spirit, House of Lore, Unsorted Bin Attack). Приведены примеры кода эксплуатации уязвимостей и классификация представленных техник в соответствии с общим перечнем Common Weakness Enumeration. Исследованы современные методы и средства обнаружения уязвимостей, показаны их достоинства и недостатки на примере работы фреймворка Heap Hopper. Рассмотрены современные подходы к верификации программного обеспечения. **Метод.** Предложенный метод верификации программного обеспечения совмещает подходы статического анализа и символьного выполнения при использовании точной модели алгоритмов выделения динамической памяти. В процессе компиляции проверяемой программы создается структура Крипке. Уязвимости динамической памяти описываются формулами темпоральной логики. Полученная структура и формулы передаются на вход алгоритма проверки моделей. Осуществляется конкретно-символьное выполнение собранного бинарного файла. Для символьных путей выполнения проверяются условия уязвимостей, выраженные в виде формул пропозициональной логики. **Основные результаты.** Показана возможность практического применения предложенного подхода к обнаружению уязвимостей, возникающих при распределении динамической памяти в программных приложениях. Символьное выполнение реализовано в виде низкоуровневого отладчика, что позволяет снизить время работы алгоритмов за счет выполнения проверяемого приложения на реальном процессоре. **Практическая значимость.** Представлен комплексный подход для решения проблемы автоматического выявления уязвимостей на разных стадиях жизненного цикла разработки программного обеспечения. Предложенный подход применим для верификации аналогичных реализаций аллокаторов динамической памяти, таких как ptmalloc, dlmalloc, tcmalloc, jemalloc, musl.

Ключевые слова
верификация, уязвимость, символьное выполнение, динамическая память, язык С

doi: 10.17586/2226-1494-2020-20-1-101-109

VERIFICATION OF DYNAMIC MEMORY ALLOCATORS BASED ON SYMBOLIC PROGRAM EXECUTION

A.M. Dergachev, D.S. Sadyrin, A.G. Ilina, I.P. Loginov, Iu.D. Korenkov

ITMO University, Saint Petersburg, 197101, Russian Federation
Corresponding author: dam600@gmail.com

Article info
Received 03.12.19, accepted 30.12.19
Article in Russian

For citation: Dergachev A.M., Sadyrin D.S., Ilina A.G., Loginov I.P., Korenkov Iu.D. Verification of dynamic memory allocators based on symbolic program execution. *Scientific and Technical Journal of Information Technologies, Mechanics and Optics*, 2020, vol. 20, no. 1, pp. 101–109 (in Russian). doi: 10.17586/2226-1494-2020-20-1-101-109

Abstract

Subject of Research. The paper presents the study of vulnerability exploitation techniques in the implementation of dynamic memory allocation algorithms (glibc library allocator): Poisoned Null-byte, Overlapped Chunks, Fastbin Attack,

Unsafe Unlink, House of Einherjar, House of Force, House of Spirit, House of Lore, Unsorted Bin Attack. Examples of vulnerability exploitation code and classification of the presented techniques are given in accordance with the Common Weakness Enumeration list. The modern methods and means of vulnerabilities detection are studied; their advantages and disadvantages are shown using the Heap Hopper framework as an example. Modern methods of appropriate software verification are considered. **Method.** The proposed software verification method combines the approaches of static analysis and symbolic execution using an accurate model of algorithms for dynamic memory allocation. In the compilation process of program being tested, the Kripke structure is created. Dynamic memory vulnerabilities are described by temporal logic formulas. The resulting structure and formulas are passed at the input of the model checking algorithm. Concrete-symbolic execution of the assembled binary file is performed. Vulnerability conditions expressed in the form of propositional logic formulas are checked for symbolic execution paths. **Main Results.** The practical use of the proposed approach to detection of dynamic memory vulnerabilities in software applications is shown. Symbolic execution is implemented in the form of a low-level debugger, which reduces the operating time of algorithms due to the execution of the application being tested on a real processor. **Practical Relevance.** The paper presents an integrated approach for solving the problem of automatic vulnerabilities detecting at different stages of the software development life cycle. This approach is applicable for verification of the similar implementations of dynamic memory allocators, such as ptmalloc, dlmalloc, tcmalloc, jemalloc and musl.

Keywords
verification, software errors, symbolic execution, dynamic memory, C language

Введение

Основным языком в сфере разработки системного программного обеспечения (ПО) является язык C, предоставляющий широкие возможности управления динамической памятью, что требует высокой квалификации разработчика. Ошибки программирования при работе с памятью, в том числе в реализациях алгоритмов выделения динамической памяти — аллокаторах, встречаются в операционных системах, например, Linux[1], языках программирования[2], браузерах[3]. Это делает потенциально уязвимым системное ПО. Эксплуатация же уязвимостей может приводить к потере или умышленному повреждению данных и прочим негативным последствиям — материальному ущербу [1]. Своевременное обнаружение уязвимостей — «каналов несанкционированного доступа к информации» [2] — необходимо для сведения к минимуму нежелательных последствий от их вредоносной эксплуатации.

Работы, ведущиеся в области этой проблемы, нацелены на поиск решений для предотвращения эксплуатации конкретных типов уязвимостей [3–7]. В связи со сложностью задачи подходы и методы к выявлению уязвимостей и защите ПО от их эксплуатации вариативны.

Среди предлагаемых решений:

— защищенный аллокатор динамической памяти, при работе которого метаданные участков памяти остаются полностью изолированными от данных приложения, а размещение объектов при выделении памяти организовано случайным образом [8];

— детектор, отслеживающий все указатели в программе и обнуляющий указатели на освобожденные участки памяти [9];

— автоматизация сбора и анализа информации о состоянии выделенной памяти и операциях над ней [10];

— автоматизация обнаружения новых способов эксплуатации динамически выделяемой памяти [11];

— верификация моделей [12].

На основе данных решений разработан ряд инструментов — HAIT [10], ArcHeap [11], MOPS [12], CBMC[4] [13], Heap Hopper [14], направленных на решение проблемы поиска ошибок и выявления уязвимостей в ПО, написанном на языке C, но ни один из них не отвечает в полной мере требованиям, предъявляемым к таким инструментам [15], а именно:

— возможность детектирования всех известных уязвимостей;

— защитные механизмы не должны вносить изменения в программный код;

— низкие накладные расходы на поиск уязвимостей;

— сохранение неизменной модели распределения динамической памяти.

Несмотря на постоянное совершенствование методов и средств обнаружения уязвимостей и внедрение защитных мер высокий уровень актуальности проблемы сохраняется. Решение проблемы — разработка стратегии автоматического выявления уязвимостей на разных стадиях жизненного цикла ПО, основанного на верификации аллокаторов динамической памяти с применением символьного выполнения программного кода.

Терминология

Задача выполнения запроса на выделение динамической памяти состоит в том, чтобы найти блок неиспользуемой памяти достаточного размера. Выполнением данной задачи занимается программа аллокатор

[1] CVE-2019-14816 in Ubuntu [Электронный ресурс]. Режим доступа: https://people.canonical.com/~ubuntu-security/cve/2019/CVE-2019-14816.html, свободный. Яз. англ. (дата обращения 20.12.2019).

[2] PHP :: Sec Bug #72293 :: Heap overflow in mysqlnd related to BIT fields [Электронный ресурс]. Режим доступа: https://bugs.php.net/bug.php?id=72293. Свободный. Яз. англ. (дата обращения 20.12.2019).

[3] Security vulnerabilities fixed in - Firefox 70 — Mozilla [Электронный ресурс]. Режим доступа: https://www.mozilla.org/en-US/security/advisories/mfsa2019-34/#CVE-2018-6156, свободный. Яз. англ. (дата обращения 20.12.2019).

[4] CBMC Homepage [Электронный ресурс]. Режим доступа: http://www.cs.cmu.edu/~modelcheck/cbmc/, свободный, Яз. англ. (дата обращения 15.01.2020).

(allocator). Запросы на выделение памяти удовлетворяются путем выделения аллокатором частей памяти нужного размера — чанков (chunk) из большого объема памяти, называемого свободным хранилищем или кучей (heap). В любой момент времени некоторые чанки используются, а некоторые являются свободными (неиспользованными) и, таким образом, доступными для аллокатора. Метаданные выделенных и свободных чанков различаются (рисунок). Для хранения метаданных используются следующие поля:

— prev_size — размер предыдущего чанка;
— size — размер текущего чанка;
— fwd — указатель на следующий чанк;
— bck — указатель на предыдущий чанк;
— AMP — поле флагов, где бит A (ARENA) показывает принадлежность чанка основному хранилищу (main arena), бит M (MMAPPED) показывает, является ли чанк выделенным с помощью mmap, бит P (PREV_INUSE) указывает на состояние предыдущего чанка в куче — используется или является свободным.

Кроме того, существует так называемый топчанк (top chunk), который представляет собой самый большой свободный чанк в куче, расположенный на верхней границе памяти, запрошенной у операционной системы. Выделение памяти из топчанка происходит лишь в том случае, если аллокатор не смог выделить чанк требуемого размера.

При освобождении чанки попадают в несортированную корзину или бин (unsorted bin) — контейнер, из которого выбираются при необходимости аллокатором и помещаются в соответствующие корзины, различающиеся по размеру хранимых чанков — быстрые бины (fast bins), малые бины (small bins) и большие бины (large bins). Корзины (бины) представляют собой односвязные или двусвязные списки свободных чанков.

Уязвимости в реализации алгоритмов выделения динамической памяти

На сегодня известно множество уязвимостей в реализации алгоритмов выделения динамической памяти в библиотеке glibc, которые подробно рассмотрены в ряде работ [16, 17] и иллюстрируются подробными примерами [18]. Как видно из табл. 1, разработчиками glibc были внесены исправления — патчи (patch), которые делают невозможным проведение ряда атак.

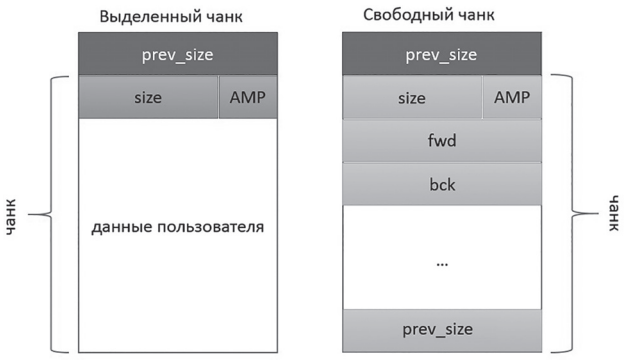

Рисунок. Метаданные свободного и выделенного чанков

Таблица 1. Наличие возможности атак для разных версий glibc

Техника	Glibc 2.25	Glibc 2.26
Poisoned Null-byte (PNB)	+	–
Overlapped Chunks (OC)	+	+
Fastbin Attack (FA)	+	+
Unsafe Unlink (UU)	+	+
House of Einherjar (HE)	+	+
House of Force (HF)	+	–
House of Spirit (HS)	+	–
House of Lore (HL)	+	+
Unsorted Bin Attack (UB)	+	+

В табл. 1 приведено сравнение версий 2.25 и 2.26 библиотеки glibc в зависимости от наличия уязвимостей.

Согласно классификации CWE (Common Weakness Enumeration)[1], атаки, представленные в табл. 1, эксплуатируют уязвимости, подпадающие под следующие категории:

— CWE-122: Heap-based Buffer Overflow;
— CWE-415: Double Free;
— CWE-416: Use After Free;
— CWE-476: NULL Pointer Dereference.

В табл. 2 показано соответствие между указанными в табл. 1 уязвимостями и техниками эксплуатации в зависимости от наличия:

— возможности перезаписи топчанка и полей prev_size, size, fwd, bck, AMP;
— возможности создания специального подложного чанка на стеке или куче;
— уязвимости Double Free;
— возможности освободить указатель по произвольному адресу (Arbitrary Free).

Проиллюстрируем поведение программ на примере техник Double Free и Unsorted Bin Attack, использующих некоторые из уязвимостей, например, CWE-122 и CWE-415.

При использовании техники Double Free в следующем коде чанк p1 освобождается два раза с занесением его в быстрый бин, после чего перезапись его поля метаданных fwd на адрес переменной stack_var позволяет последующим вызовом malloc вернуть чанк pp4 по произвольному адресу, в данном случае по адресу (char *)&stack_var+8, что отражено в выводе программы.

```
unsigned long long stack_var;
fprintf(stderr, «The address we want malloc() to return is
%p.\n», 8+(char *)&stack_var);
unsigned long long *p1 = malloc(8);
unsigned long long *p2 = malloc(8);
free(p1);
free(p2);
free(p1);
```

[1] Common Weakness Enumeration [Электронный ресурс]. Режим доступа: https://cwe.mitre.org/index.html, свободный. Яз. англ. (дата обращения 15.01.2020).

Таблица 2. Соответствие уязвимостей и техник эксплуатации

Техника	Требует наличия возможности перезаписи области памяти или полей метаданных						Требует создания подложного чанка		Требует наличия Double Free	Требует наличия Arbitrary Free
	Топчанк	prev_size	AMP	size	fwd	bck	На стеке	На куче		
PNB			+							
OC				+						
FA					+				+	
UU		+	+					+		
HE		+	+	+						
HF	+									
HS							+			+
HL						+	+			
UB		+	+	+		+				

```
unsigned long long *pp1 = malloc(8);
unsigned long long *pp2 = malloc(8);
stack_var = 0x20;
*pp1 = (unsigned long long) (((char*)&stack_var) — sizeof(pp1));
unsigned long long *pp3 = malloc(8);
unsigned long long *pp4 = malloc(8);
fprintf(stderr, «allocated chunk: %p, %p, %p, %p\n», pp1, pp2, pp3, pp4);
```

Вывод программы:
The address we want malloc() to return is 0x7ffc4099d018.
allocated chunk: 0x1edb010, 0x1edb030, 0x1edb010, 0x7ffc4099d018

В случае техники Unsorted Bin Attack в приведенном ниже примере чанк p при освобождении заносится в несортированный бин. Производится перезапись значения поля bck у освобожденного чанка на адрес переменной, расположенной на стеке. Далее чанк забирается из бина. В процессе изымания чанка из начала несортированного бина, при перезаписи значений указателей fwd на bck, а bck на fwd происходит запись значения служебной константы unsorted_chunks (av) по указанному в поле bck адресу переменной в стеке программы.

```
unsigned long stack_var=0;
fprintf(stderr, “target we want to rewrite on stack:\n”);
fprintf(stderr, “%p: %ld\n\n”, &stack_var, stack_var);
unsigned long *p=malloc(400);
malloc(500);
free(p);
p[1]=(unsigned long)(&stack_var-2);
malloc(400);
fprintf(stderr, “target rewritten:\n”);
fprintf(stderr, «%p: %p\n», &stack_var, (void*)stack_var);
```

Вывод программы:
target we want to rewrite on stack:
0x7ffe01edd908: 0
target rewritten:
0x7ffe01edd908: 0x7f3d747efc58

Очевидно, что возникновение аналогичных ситуаций при эксплуатации ПО недопустимо, что приводит к неизбежному поиску путей решения задачи автоматизации борьбы с рассмотренными явлениями на всех этапах жизненного цикла ПО.

Символьное выполнение программ

Существуют две группы подходов к аналитике качества программ — методы статического анализа и методы динамического анализа. Из методов динамического анализа наиболее перспективным в рамках поставленной задачи представляется символьное выполнение (symbolic execution) [19] программы. Символьное выполнение относится к формальным методам верификации. Данная техника позволяет проводить моделирование выполнения программы, при котором часть входных переменных представляется в символьном виде. Символ переменной обозначает множество значений входной переменной программы из области ее определения. Каждое символьное выполнение эквивалентно выполнению программы на наборе конкретных тестовых значений входных переменных, что сокращает мощность множества создаваемых тестов. От символьного выполнения требуется подобрать входные данные, на которых произойдет ошибка. Генерация наборов входных данных для исследования путей выполнения программы реализуется по следующему сценарию:

— пути выполнения программы связываются с входными данными набором булевых формул;

— для конкретного пути выполнения программы строится трасса ограничений;

— сравнение исследуемого пути выполнения и некоторого желаемого пути выполнения с последующей инструментальной проверкой булевых формул позволяет сгенерировать набор входных данных, который приведет к выполнению программы по желаемому пути.

Следует отметить, что применение метода подразумевает накладные расходы на выполнение программы, которые в некоторых случаях могут составлять до 500 %, что объясняется высокой вычислительной сложностью данного подхода [20]. В табл. 3 демонстрируется пример символьного исполнения фрагмента программного кода.

Таблица 3. Пример символьного исполнения

Исходный код	Символьное исполнение	
1: int x; 2: x = getc(); 3: if (x < 12);	1: завести символьную переменную sym _x над доменом всех значений типа int 2: завести символьную переменную sym _stdin_1 над доменом всех значений типа возвращаемого значения функции getc() – символа со стандартного потока ввода. Составить ограничение (sym_x == sym_stdin_1) 3: создать две независимые ветки исполнения	
4: do1 5: else 6: do2	3: составить ограничение (sym_x<12) 4: продолжить символьное исполнение по блоку do1	5: составить ограничение (sym_x>=12) 6: продолжить символьное исполнение по блоку do2

Из рассмотренных в ходе исследования инструментах верификации, наиболее подходящим оказался Heap Hopper [14], реализованный на фреймворке Angr [21]. Одним из достоинств фреймворка Angr является то, что в нем, помимо символьного исполнения кода приложения, доступно также символьное выполнение кода из разделяемых библиотек. Недостатком фреймворка Angr является то, что он выполнен в виде эмулятора, что приводит к снижению скорости символьного выполнения машинных инструкций.

Основные составляющие фреймворка Angr обладают следующими функциями:

— claripy — решение условий достижимости;

— PyVEX — перевод машинных инструкций в абстрактное представление;

— cle — выполнение бинарного кода;

— archinfo — хранение архитектурно-зависимой информации.

Анализ кода в Heap Hopper реализован следующим образом. На каждом шаге выполнения программы создается объект класса SimState, в котором хранится состояние регистров и памяти программы в данный момент. Регистры и память могут иметь конкретное, либо символьное значение. Каждая символьная переменная представляется в виде класса BitVectorSymbol. Также существует возможность вручную помечать необходимые входные данные как символьные — это может быть символьная память, представленная в виде класса SimSymbolicMemory, или символьный файл, представленный классом SimFile. По достижении инструкции условного перехода добавляется ограничение на символьную переменную.

При вызове функции malloc с символьным параметром создается чанк памяти с символьными метаданными, и возвращается символьный адрес памяти, а в классе SimState сохраняется состояние кучи. Символьные адреса сохраняются в полях malloc_dict, free_dict объекта класса HeapConditionTracker. После каждого вызова malloc производится проверка условий для разных типов уязвимостей, таких как arbitrary write, overlapped allocations, non-heap address allocation, non-heap address free, double free.

Для каждой рассмотренной в табл. 1 техники составляются тестовые примеры с уязвимым кодом. В примерах отсутствуют значения переменных, таких как: size_t malloc_sizes[11], size_t fill_sizes[11], size_t header_size. Память исполняемого файла, соответствующая данным переменным, помечается как символьная, их значения вычисляются в ходе символьного выполнения на основе проверки условий для уязвимостей, т. е. выбираются только те символьные значения, которые приводят состояние программы к уязвимости некоторого типа. Код примера с вычисленными константами компилируется в исполняемый файл. Вызов функции read заменяется записью в память. Это дает возможность проверить применимость техник для различных версий аллокатора glibc независимо от изменений, внесенных в код. Ниже приведен тестовый пример для техники Unsorted Bin Attack.

Тестовый пример для техники Unsorted Bin Attack

```
...
size_t write_target[4];
size_t offset;
size_t header_size;
size_t mem2chunk_offset;
size_t malloc_sizes[3];
size_t fill_sizes[3];
...
int main(void) {
	void *dummy_chunk = malloc(0x200);
	free(dummy_chunk);

	ctrl_data_0.global_var = malloc(malloc_sizes[0]);
	for (int i=0; i < fill_sizes[0]; i+=8) {
		read(0, ((uint8_t *)ctrl_data_0.global_var)+i, 8);
	}

	ctrl_data_1.global_var = malloc(malloc_sizes[1]);
	for (int i=0; i < fill_sizes[1]; i+=8) {
		read(0, ((uint8_t *)ctrl_data_1.global_var)+i, 8);
	}

	free(ctrl_data_0.global_var);
	read(3, ctrl_data_0.global_var, header_size); // <--- VULN
	ctrl_data_1.global_var = malloc(malloc_sizes[2]);
	for (int i=0; i < fill_sizes[2]; i+=8) {
		read(0, ((uint8_t *)ctrl_data_2.global_var)+i, 8);
	}
}
```

Тестовый пример после символьного выполнения

```
...
size_t write_target[4];
size_t offset;
size_t header_size = 0x20;
size_t mem2chunk_offset = 0x10;
size_t malloc_sizes[3] = { 0x190L, 0x190L, 0x190L };
size_t fill_sizes[3] = { 0x0L, 0x0L, 0x0L };
```

```
...
int main(void){
        ...
        //VULN
        ctrl_data_0.global_var[0] = 0x0;
       ctrl_data_0.global_var[1] = ((char *) &write_target) + 0x5;
        ctrl_data_0.global_var[2] = 0x0;
        ctrl_data_0.global_var[3] = 0x0;

        write_target[0] = (uint64_t) 0x0;
        write_target[1] = (uint64_t) 0x0;
        write_target[2] = (uint64_t) 0x0;
        write_target[3] = (uint64_t) 0x0;
        for (int i = 0; i < 4; i++) {
                printf("write_target[%d]: %p\n", i, (void *)
                write_target[i]);
        }

        ctrl_data_1.global_var = malloc(malloc_sizes[2]);
        printf("Allocation: %p\nSize: 0x%lx\n",
        ctrl_data_1.global_var, malloc_sizes[2]);

        for (int i = 0; i < 4; i++) {
                printf("write_target[%d]: %p\n", i, (void *)
                write_target[i]);
        }
}
```

```
Результат выполнения тестового примера
Allocation: 0x7f3bdf6b9420
Size: 0x190
Allocation: 0x7f3bdf6b95c0
Size: 0x190
Free: 0x7f3bdf6b9420
write_target[0]: (nil)
write_target[1]: (nil)
write_target[2]: (nil)
write_target[3]: (nil)
Allocation: 0x7f3bdf6b9420
Size: 0x190
write_target[0]: (nil)
write_target[1]: (nil)
write_target[2]: 0x146b780000000000
write_target[3]: 0x7f3bdf
```

Данный подход также применим к другим аллокаторам памяти, так как различия во внутреннем устройстве аллокаторов (метаданные, число бинов, размер чанков в бинах) учитываются при генерации примеров в ходе символьного выполнения. Ниже перечислены наиболее известные аллокаторы:

— ptmalloc — используется в операционных системах Ubuntu, Debian, Fedora;

— tcmalloc — используется в Google Chrome;

— jemalloc — используется в Firefox, в операционной системе FreeBSD;

— musl — используется в альтернативной реализации glibc (musl libc);

— Hoard — эффективен при использовании многопоточных приложений.

Анализ алгоритмов инструмента Heap Hopper позволил выяснить, что при выполнении malloc не учитывается внутреннее состояние кучи, представленное структурой malloc_state — состояние полей bins, fastbins, binmap, top[1]. Тем самым, можно считать, что модель кучи, представленная в Heap Hopper, является в значительной степени приближенной, что создает перспективы для усовершенствования данного подхода.

Предлагаемый подход

Решение задачи поиска ошибок в коде программ, связанных с уязвимостями в алгоритмах распределения динамической памяти, несмотря на ряд существующих инструментов, остается актуальным. Предлагаемый авторами комплексный подход основан на совмещении статического анализа и символьного выполнения, что позволяет разработать инструмент, выявляющий в программном коде все известные уязвимости аллокаторов, снизив при этом накладные расходы по сравнению с уже существующими инструментами за счет выполнения проверяемого приложения на реальном процессоре.

Разрабатываемый на основе предлагаемого подхода инструмент представляет собой программу-отладчик. Входными данными для программы являются бинарные файлы проверяемого приложения.

Принцип работы отладчика и заложенного в него предлагаемого подхода заключается в следующем. В первую очередь на основании бинарных файлов и подробной модели алгоритмов выделения памяти производится моделирование распределения динамической памяти в программе. Далее осуществляется динамическое символьное выполнение исполняемого файла, также известное как конкретно-символьное (concolic — concrete and symbolic), которое совмещает реальное и символьное выполнение программы. Динамическое символьное выполнение позволяет применять техники исследования путей программы и, добавляя предикаты безопасности к ограничениям пути (path constraint), проверять потенциально опасные операции на наличие реальных ошибок в программах, например:

— Allocated chunks overlap

$\exists B$: $((A < B) \wedge (A + sizeof(A) > B)) \vee ((A > B) \wedge (B + sizeof(B) > A))$;

— Bad pointer free

$(ptr < heap\_location) \vee (ptr \in freed\_ptr\_array)$.

Для каждого состояния символьного выполнения проверяется ряд условий:

— возможность атакующего по перезаписи топчанка и полей метаданных;

— возможность создать подложный чанк на стеке или куче;

— наличие уязвимости Double Free;

— возможность освободить указатель по произвольному адресу (Arbitrary free).

В процессе символьного выполнения для выделенных с учетом указанных условий путей выполнения программы производится генерация наборов символьных переменных как для данных программы, так и для метаданных чанков. Для этого производится ре-

[1] Understanding glibc malloc [Электронный ресурс]. Режим доступа: https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/comment-page-1/, свободный. Яз. англ. (дата обращения 15.01.2020).

шение задачи выполнимости булевых формул (boolean satisfiability problem).

Поскольку отладчик разрабатывается с элементами визуализации, на каждом ветвлении может быть легко проанализирована картина выявленных уязвимостей. На выходе программы формируется отчет по каждой выявленной уязвимости и наборах символьных данных, приводящих к возможности ее эксплуатации.

Для поиска некоторых уязвимостей (Double Free, Use-After-Free) нет необходимости учитывать метаданные чанков, достаточно учитывать только выполненные в программе операции, поэтому можно применять методы статического анализа кода. При наличии исходного кода проверяемого приложения выполняется построение графа потока управления (control flow graph), вершины помечаются синтаксическими конструкциями из дерева синтаксического разбора. Полученный граф вместе с метками на вершинах рассматривается как структура Крипке [22]. Для каждой уязвимости составляется темпоральная формула CTL (Computation Tree Logic) с использованием меток на вершинах графа.

Темпоральная логика описывает свойства всех путей развития системы во времени. Для описания формул логики CTL используются следующие операторы:

— AG — для всех путей в любой момент времени;

— AF — для всех путей в некоторый момент времени;

— AX — для всех путей в следующий момент времени;

— EG — любой момент времени для какого-либо пути;

— EF — для какого-либо пути в некоторый момент времени;

— EX — для какого-либо пути в следующий момент времени.

Например, для уязвимости Use-After-Free CTL-формула будет выглядеть следующим образом: $AG(malloc_p \rightarrow AG(free_p \rightarrow not(EF\ used_p)))$, т. е. для всех путей, для которых после вызова malloc есть вызов free для ресурса p, не существует такого пути, чтобы p использовался позже вызова free.

Структура Крипке и CTL-формулы передаются на вход алгоритму проверки моделей [23], который в случае, если одна из формул нарушается, генерирует контрпример.

Для построения более точного представления кучи в процессе символьного выполнения работа ведется напрямую со структурой malloc_state, и используется фреймворк Triton [24]. В данном фреймворке уже реализовано символьное выполнение и тейнт-анализ (taint checking), что позволяет сосредоточиться непосредственно на разработке и отладке алгоритмов поиска атак на динамическую память.

Подход статического анализа и динамического символьного выполнения предполагается совместить при разработке дополнительного модуля корректировки проверяемого кода на этапе компиляции с целью введения препятствий для эксплуатации уязвимостей, обнаруженных в процессе анализа программного кода.

Заключение

В работе рассмотрены известные техники атак на динамическую память, подход к верификации программного обеспечения на основе методов символьного выполнения, кратко охарактеризован существующий инструмент Heap Hopper. Предложен альтернативный подход выявления уязвимостей в аллокаторах динамической памяти с использованием динамического символьного выполнения, а также комплексный подход, совмещающий статический анализ и символьное выполнение программного кода. В дальнейшем данный подход планируется распространить на другие известные реализации аллокаторов динамической памяти.

Литература

1. Андреев Ю.С., Дергачев А.М., Жаров Ф.А., Садырин Д.С. Информационная безопасность автоматизированных систем управления технологическими процессами // Известия высших учебных заведений. Приборостроение. 2019. Т. 62. № 4. С. 331–339. doi: 10.17586/0021-3454-2019-62-4-331-339
2. Щеглов К.А., Щеглов А.Ю. Эксплуатационные характеристики риска нарушений безопасности информационной системы // Научно-технический вестник информационных технологий, механики и оптики. 2014. № 1(89). С. 129–139.
3. Repel D., Kinder J., Cavallaro L. Modular synthesis of heap exploits // Proc. of the 2017 Workshop on Programming Languages and Analysis for Security, PLAS'17. 2017. P. 25–35. doi: 10.1145/3139337.3139346
4. Heelan S., Melham T., Kroening D. Automatic heap layout manipulation for exploitation // Proc. 27th USENIX Security Symposium. 2018. P. 763–779.
5. Liu D., Zhang M., Wang H. A robust and efficient defense against use-after-free exploits via concurrent pointer sweeping // Proc. 25th ACM Conference on Computer and Communications Security (CCS 2018). 2018. P. 1635-1648. doi: 10.1145/3243734.3243826
6. Nikiforakis N., Piessens F., Joosen W. HeapSentry: Kernel-assisted protection against heap overflows // Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). 2013. V. 7967. P. 177–196. doi: 10.1007/978-3-642-39235-1_11

References

1. Andreev Yu.S., Dergachev A.M., Zharov F.A., Sadyrin D.S. Information security of automated control systems of technological processes. *Journal of Instrument Engineering*, 2019, vol. 62, no. 4, pp. 331–339. (in Russian). doi: 10.17586/0021-3454-2019-62-4-331-339
2. Shcheglov K., Shcheglov A. Operational characteristics of information system security threats risk. *Scientific and Technical Journal of Information Technologies, Mechanics and Optics*, 2014, no. 1(89), pp. 129–139. (in Russian)
3. Repel D., Kinder J., Cavallaro L. Modular synthesis of heap exploits. *Proc. of the 2017 Workshop on Programming Languages and Analysis for Security, PLAS'17*, 2017, pp. 25–35. doi: 10.1145/3139337.3139346
4. Heelan S., Melham T., Kroening D. Automatic heap layout manipulation for exploitation. *Proc. 27th USENIX Security Symposium*, 2018, pp. 763–779.
5. Liu D., Zhang M., Wang H. A robust and efficient defense against use-after-free exploits via concurrent pointer sweeping. *Proc. 25th ACM Conference on Computer and Communications Security (CCS 2018)*, 2018, pp. 1635-1648. doi: 10.1145/3243734.3243826
6. Nikiforakis N., Piessens F., Joosen W. HeapSentry: Kernel-assisted protection against heap overflows. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2013, vol. 7967, pp. 177–196. doi: 10.1007/978-3-642-39235-1_11

7. Lee B., Song C., Jang Y., Wang T., Kim T., Lu L., Lee W. Preventing use-after-free with dangling pointers nullification [Электронный ресурс]. URL: https://wenke.gtisc.gatech.edu/papers/dangnull.pdf, свободный. Яз. англ. (дата обращения: 15.01.2020). doi: 10.14722/ndss.2015.23238
8. Novark G., Berger E.D. DieHarder: Securing the heap // Proc. 17th ACM Conference on Computer and Communications Security, CCS'10. 2010. P. 573–584. doi: 10.1145/1866307.1866371
9. Caballero J., Grieco G., Marron M., Nappa A. Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities // Proc. 21st International Symposium on Software Testing and Analysis (ISSTA 2012). 2012. P. 133–143. doi: 10.1145/04000800.2336769
10. Atzeni A., Marcelli A., Muroni F., Squillero G. HAIT: Heap analyzer with input tracing // ICETE 2017 – Proc. 14th International Joint Conference on e-Business and Telecommunications. 2017. V. 4. P. 327–334. doi: 10.5220/0006420803270334
11. Yun I., Karil D., Kim T. Automatic techniques to systematically discover new heap exploitation primitives [Электронный ресурс]. URL: https://arxiv.org/pdf/1903.00503.pdf, свободный. Яз. англ. (дата обращения: 15.01.2020).
12. Chen H., Wagner D. MOPS: An infrastructure for examining security properties of software // Proc. 9th ACM Conference on Computer and Communications Security. 2002. P. 235–244. doi: 10.1145/586110.586142
13. Clarke E., Kroening D., Lerda F. A tool for checking ANSI-C programs // Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). 2004. V. 2988. P. 168–176. doi: 10.1007/978-3-540-24730-2_15
14. Eckert M, Bianchi A, Wang R, Shoshitaishvili Y, Kruegel C, Vigna G. HeapHopper: Bringing bounded model checking to heap implementation security // Proc. 27th USENIX Security Symposium. 2018. P. 99–116.
15. Xu W., DuVarney D.C., Sekar R. An efficient and backwards-compatible transformation to ensure memory safety of C programs // Proc. Twelfth ACM SIGSOFT International Symposium on the Foundations of Software Engineering, SIGSOFT 2004/FSE-12. 2004. P. 117–126. doi: 10.1145/1029894.1029913
16. Yet another free() exploitation technique [Электронный ресурс]. URL: http://phrack.org/issues/66/6.html, свободный. Яз. англ. (дата обращения: 15.01.2020).
17. Malloc Des-Maleficarum [Электронный ресурс]. URL: http://www.phrack.org/issues/66/10.html, свободный. Яз. англ. (дата обращения: 15.01.2020).
18. A repository for learning various heap exploitation techniques [Электронный ресурс]. URL: https://github.com/shellphish/how2heap, Яз. англ. (дата обращения: 15.01.2020).
19. Dudina I.A., Belevantsev A.A. Using static symbolic execution to detect buffer overflows // Programming and Computer Software. 2017. V. 43. N 5. P. 277–288. doi: 10.1134/S0361768817050024
20. Ермаков М.К. Методы повышения эффективности итеративного динамического анализа программ: диссертация на соискание ученой степени кандидата математических наук. М.: Московский государственный университет им. М.В. Ломоносова, 2016.
21. Wang F., Yan S. Angr - the next generation of binary analysis // Proc. IEEE Cybersecurity Development Conference (SecDev 2017). 2017. P. 8–9. doi: 10.1109/SecDev.2017.14
22. Kripke structure (model checking) [Электронный ресурс]. URL: https://en.wikipedia.org/wiki/Kripke_structure_(model_checking) свободный. Яз. англ. (дата обращения: 15.01.2020).
23. Bhat G., Cleaveland R., Grumberg O. Efficient on-the-fly model checking for CTL // Proc. 10th Annual IEEE Symposium on Logic in Computer Science. 1995. P. 388–397. doi: 10.1109/LICS.1995.523273
24. Saudel F., Salwan J. Triton: A dynamic symbolic execution framework // Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes. 2015. P. 31–54.

7. Lee B., Song C., Jang Y., Wang T., Kim T., Lu L., Lee W. *Preventing use-after-free with dangling pointers nullification*. Available at: https://wenke.gtisc.gatech.edu/papers/dangnull.pdf (accessed: 15.01.2020). doi: 10.14722/ndss.2015.23238
8. Novark G., Berger E.D. DieHarder: Securing the heap. *Proc. 17th ACM Conference on Computer and Communications Security*, *CCS'10*, 2010, pp. 573–584. doi: 10.1145/1866307.1866371
9. Caballero J., Grieco G., Marron M., Nappa A. Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities. *Proc. 21st International Symposium on Software Testing and Analysis (ISSTA 2012)*, 2012, pp. 133–143. doi: 10.1145/04000800.2336769
10. Atzeni A., Marcelli A., Muroni F., Squillero G. HAIT: Heap analyzer with input tracing. *ICETE 2017 — Proc. 14th International Joint Conference on e-Business and Telecommunications*, 2017, vol. 4, pp. 327–334. doi: 10.5220/0006420803270334
11. Yun I., Karil D., Kim T. *Automatic techniques to systematically discover new heap exploitation primitives*. Available at: https://arxiv.org/pdf/1903.00503.pdf (accessed: 15.01.2020).
12. Chen H., Wagner D. MOPS: An infrastructure for examining security properties of software. *Proc. 9th ACM Conference on Computer and Communications Security*, 2002, pp. 235–244. doi: 10.1145/586110.586142
13. Clarke E., Kroening D., Lerda F. A tool for checking ANSI-C programs. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2004, vol. 2988, pp. 168–176. doi: 10.1007/978-3-540-24730-2_15
14. Eckert M, Bianchi A, Wang R, Shoshitaishvili Y, Kruegel C, Vigna G. HeapHopper: Bringing bounded model checking to heap implementation security. *Proc. 27th USENIX Security Symposium*, 2018, pp. 99–116.
15. Xu W., DuVarney D.C., Sekar R. An efficient and backwards-compatible transformation to ensure memory safety of C programs. *Proc. Twelfth ACM SIGSOFT International Symposium on the Foundations of Software Engineering, SIGSOFT 2004/FSE-12*, 2004, pp. 117–126. doi: 10.1145/1029894.1029913
16. *Yet another free() exploitation technique*. Available at: http://phrack.org/issues/66/6.html (accessed: 15.01.2020).
17. *Malloc Des-Maleficarum*. Available at: http://www.phrack.org/issues/66/10.html (accessed: 15.01.2020).
18. *A repository for learning various heap exploitation techniques*. Available at: https://github.com/shellphish/how2heap (accessed: 15.01.2020).
19. Dudina I.A., Belevantsev A.A. Using static symbolic execution to detect buffer overflows. *Programming and Computer Software*, 2017, vol. 43, no. 5, pp. 277–288. doi: 10.1134/S0361768817050024
20. Ermakov M.K. *Efficiency Improvement of Iterative Dynamic Program Analysis*. Dissertation for the degree of candidate of mathematical sciences. Moscow, Lomonosov Moscow State University, 2016. (in Russian)
21. Wang F., Yan S. Angr — the next generation of binary analysis. *Proc. IEEE Cybersecurity Development Conference (SecDev 2017)*, 2017, pp. 8–9. doi: 10.1109/SecDev.2017.14
22. *Kripke structure (model checking)*. Available at: https://en.wikipedia.org/wiki/Kripke_structure_(model_checking) (accessed: 15.01.2020).
23. Bhat G., Cleaveland R., Grumberg O. Efficient on-the-fly model checking for CTL. *Proc. 10th Annual IEEE Symposium on Logic in Computer Science*, 1995, pp. 388–397. doi: 10.1109/LICS.1995.523273
24. Saudel F., Salwan J. Triton: A dynamic symbolic execution framework. *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC*, France, Rennes, 2015, P. 31–54.

Авторы

Дергачев Андрей Михайлович — кандидат технических наук, доцент, Университет ИТМО, Санкт-Петербург, 197101, Российская Федерация, Scopus ID: 57205166082, ORCID ID: 0000-0002-1754-7120, dam600@gmail.com
Садырин Даниил Сергеевич — аспирант, Университет ИТМО, Санкт-Петербург, 197101, Российская Федерация, Scopus ID: 57211288674, ORCID ID: 0000-0001-5002-3639, dssadyrin@itmo.ru
Ильина Аглая Геннадьевна — кандидат технических наук, доцент, Университет ИТМО, Санкт-Петербург, 197101, Российская Федерация, ORCID ID: 0000-0003-1866-7914, agilina@itmo.ru
Логинов Иван Павлович — ассистент, Университет ИТМО, Санкт-Петербург, 197101, Российская Федерация, Scopus ID: 56719519200, ORCID ID: 0000-0002-6254-6098, ivan.p.loginov@gmail.com
Кореньков Юрий Дмитриевич — ассистент, Университет ИТМО, Санкт-Петербург, 197101, Российская Федерация, Scopus ID: 57205163047, ORCID ID: 0000-0002-8948-2776, ged.yuko@gmail.com

Authors

Andrey M. Dergachev — PhD, Associate Professor, ITMO University, Saint Petersburg, 197101, Russian Federation, Scopus ID: 57205166082, ORCID ID: 0000-0002-1754-7120, dam600@gmail.com
Daniil S. Sadyrin — Postgraduate, ITMO University, Saint Petersburg, 197101, Russian Federation, Scopus ID: 57211288674, ORCID ID: 0000-0001-5002-3639, dssadyrin@itmo.ru
Aglaya G. Ilina — PhD, Associate Professor, ITMO University, Saint Petersburg, 197101, Russian Federation, ORCID ID: 0000-0003-1866-7914, agilina@itmo.ru,
Ivan P. Loginov — Assistant, ITMO University, Saint Petersburg, 197101, Russian Federation, Scopus ID: 56719519200, ORCID ID: 0000-0002-6254-6098, ivan.p.loginov@gmail.com
Iurii D. Korenkov — Assistant, ITMO University, Saint Petersburg, 197101, Russian Federation, Scopus ID: 57205163047, ORCID ID: 0000-0002-8948-2776, ged.yuko@gmail.com