خلاصه‌ی مقاله‌ی «تشخیص دقیق و مقیاس‌پذیر اِشکال واکشی مضاعف در هسته‌ی سیستم‌عامل[1]»

در مقاله‌ی مذکور به تشخیص اتوماتیک اشکال واکشی مضاعف در هسته‌ی لینوکس و FreeBSD با استفاده از ابزار DEADLINE که توسط نویسندگان مقاله طراحی و ساخته شده پرداخته شده است. این مقاله ابتدا تعریفی دقیق و رسمی از اشکال واکشی مضاعف ارائه داده و میان آن و خواندن چندباره[2] تمایز قائل می‌شود. به عبارتی هر اشکال واکشی مضاعف یک خواندن چندباره است اما عکس آن صحیح نیست. تعریف رسمی‌ای که در این مقاله از اشکال مذکور ارائه شده است به شرح زیر است:

۱) باید حتما خواندن دوبار و یا بیشتر از حافظه‌ی فضای کاربر[3] اتفاق افتاده باشد (حتما خواندن چندباره است).
۲) باید میان دو حافظه‌ی خوانده شده هم‌پوشانی وجود داشته باشد.
۳) باید حداقل یکی از وابستگی‌های داده‌ای و یا کنترلی بین دو بار واکشی از فضای کاربر در مورد فضای مورد هم‌پوشانی برقرار باشد.
۴) باید نتوان ثابت کرد که وابستگی مورد اشاره پس از واکشی دوم نیز هنوز برقرار است.

مثالی از وابستگی کنترلی اعمال sanity check بر روی داده است به صورتی که اگر مقدار آن از نظر هسته‌ی سیستم‌عامل معتبر نبود، عمل فراخوانی سیستمی را متوقف کند. مثالی از وابستگی داده‌ای ابتدا خواندن متغیری مانند size از فضای کاربر و ذخیره‌ی آن در فضای هسته و اعتبارسنجی آن و سپس دریافت تمامی داده از فضای کاربر است. در صورتی وابستگی داده‌ای در این مورد برقرار است که size در فضای هم‌پوشانی دو بار خواندن قرار بگیرد.

این اشکال از آن جهت اهمیت دارد که اگر برنامه‌ی کاربر چندنخی بوده و میان دو عمل واکشی بتواند داده‌ی موجود در قسمت هم‌پوشانی میان دو واکشی را دستکاری کند آنگاه بسته به کد هسته و میزان Exploitability آن می‌تواند مخرب باشد. این حمله یکی از حملات race condition است که در آن هسته برای خواندن و نخ سطح کاربر برای نوشتن حافظه رقابت دارند. در ادامه به الگوریتمی که DEADLINE طبق آن عمل می‌کند پرداخته شده است.

الگوریتم۱ دید مجرد از الگوریتم تشخیص واکشی مضاعف

```
   In  : Kernel - The kernel to be checked
   Out : Bugs - The set of double-fetch bugs found
1  Bugs ← ∅
2  Set_f ← Collect_Fetches(Kernel);
3  for F ∈ Set_f do
4      Set_mr ← Collect_Multi_Reads(F)
5      for < F_0, F_1, Fn >∈ Set_mr do
6          Paths ← Construct_Execution_Paths(F_0, F_1, Fn)
7          for P ∈ Paths do
8              if Symbolic_Checking(P, F_0, F_1) == UNSAFE then
9                  Bugs.add(< F_0, F_1 >)
10             end
11         end
12     end
13 end
```

این ابزار ابتدا تمامی واکشی‌های موجود در کد کرنل را در یک مجموعه گردآوری می‌کند. این عمل با توجه به این واقعیت انجام می‌شود که تعداد محدودی تابع برای خواندن از فضای کاربر در هسته‌ی سیستم‌عامل موجود هستند (به دلیل مدیریت بهتر مرز میان این دو فضای حافظه‌ای). بدین ترتیب قدم اول به سادگی قابل انجام است. سپس به ازای هر واکشی، تمامی خواندن‌های چندباره‌ای که این واکشی در آن دخیل است به صورت زوج

[1] Precise and Scalable Detection of Double-Fetch Bugs in OS Kernels
[2] Multi-reads
[3] Userspace Memory

واکشی‌ها در مجموعه‌ی دیگری (Set_{mr}) ذخیره می‌شود. نحوه‌ی انجام این عمل در الگوریتم۲ توضیح داده شده است. سپس مسیرهای اجرایی که از واکشی اول به دوم موجود هستند در مجموعه‌ی مسیرها قرار داده شده و به ازای هر مسیر یک بررسی سمبولیک بر روی آن اعمال می‌شود تا شروط گفته شده در تعریف اشکال در قسمت قبل را درباره‌ی آن مسیر بررسی کند.

الگوریتم۲ استخراج زوج واکشی‌های خواندن چندباره

```
   In  : F - A fetch, i.e., a call to a transfer function
   Out : R - A set of triples < F0, F1, Fn > representing multi-reads
 1 Fn ← Function that contains F;
 2 R ← ∅;
 3 Set_up ← Get_Upstream_Instructions(Fn, F);
 4 for I ∈ Set_up do
 5 |   if I is a fetch then
 6 |   |   R.add(< I, F, Fn >)
 7 |   end
 8 |   if I is a call to a function that contains a fetch then
 9 |   |   Inline I, redo the algorithm
10 |   end
11 end
12 Set_dn ← Get_Downstream_Instructions(Fn, F);
13 for I ∈ Set_dn do
14 |   if I is a fetch then
15 |   |   R.add(< F, I, Fn >)
16 |   end
17 |   if I is a call to a function that contains a fetch then
18 |   |   Inline I, redo the algorithm
19 |   end
20 end
```

برای استخراج زوج واکشی‌هایی که با واکشی F تشکیل خواندن چندباره می‌دهند، نیاز به دانستن تابعی داریم که F در آن فراخوانی شده است. سپس F را در این تابع مرجع قرار داده و دستورات بالاتر و پایین‌تر از آن را بررسی می‌کنیم. اگر دستور واکشی بود، آن را به مجموعه‌ی خواندن چندباره اضافه می‌کنیم و اگر تابعی حاوی واکشی بود، آن را باز کرده و دوباره الگوریتم را تکرار می‌کنیم.

برای یافتن مسیرهای بین دو واکشی نیز از پیمایش گراف جریان کنترلی استفاده می‌شود. در این مرحله دستوراتی که تأثیرگذار بر واکشی و یا تأثیرپذیر از آن نیستند، حذف می‌شوند. دستوری از واکشی تأثیر می‌گیرد که از مقادیر واکشی شده مشتق شده باشد و یا مقدار واکشی شده را محدود کند و دستوری تأثیر می‌گذارد که آدرس و یا سایز واکشی از آن مشتق شده و یا به واسطه‌ی آن محدود شده باشد.

در انتها با تشکیل یک نمایش سمبولیک از روی خروجی LLVM کد مربوط به مسیر میان واکشی‌ها، این نمایش سمبولیک به یک SMT[4] Solver داده می‌شود تا وجود یا عدم وجود اشکال را بررسی کند.

با اعمال این تحلیل ایستا به کرنل لینوکس و FreeBSD به ترتیب ۲۳ و ۱ اشکال جدید در کد آن‌ها کشف شد. این اشکالات به دلایل مختلفی با روش‌های قبلی قابل تشخیص نبوده است. از آن دلایل می‌توان به تشخیص روش‌های قبلی با استفاده از تحلیلگرهای لغوی (با یافتن الگوی مشخص در کد) اشاره کرد در حالی که تمامی موارد رخداد اشکال در یک یا چند الگوی خاص قابل گنجانده شدن نیست.

از محدودیت‌های این روش می‌توان به ۱) کدهای غیرقابل تبدیل به LLVM ۲) محدودیت ۴۰۹۶ مسیر مختلف میان دو واکشی و ۳) نادیده گرفتن inline assembly اشاره کرد.

[4] Satisfiability Modulo Theories

مراجع

[1] M. Xu, C. Qian, K. Lu, M. Backes and T. Kim, "Precise and Scalable Detection of Double-Fetch Bugs in OS Kernels," in *IEEE Symposium on Security and Privacy*, San Francisco, CA, US, 2018.