Zabezpečovacie techniky na úrovni prekladačov

Dávid Bolvanský

2. apríla 2019

Abstrakt

Tento článok približuje najnovšie bezpečné hrozby ako sú Spectre a Meltdown a skúma možnosti prekladača ako prostriedku na obranu proti týmto hrozbám. Dôraz je kladený na vysvetlenie princípu jednotlivých zraniteľností, možnostiam obrany proti nim a spôsobom implementácie obranných techník, tzv. *mitigations*. Ďalej skúmame dopad týchto opatrení na výkon samotných programov. Jadrom článku je analýza úlohy prekladačov v boji proti týmto hrozbách. Prezentované sú opatrenia implementované v najznámejších prekladačoch na obranu proti známym zraniteľnostiam. V závere článku prebieha diskusia a zhodnotenie týchto opatrení. Prezentovaný je návrh nových opatrení na úrovni prekladača a budúcnosť smerovania vývoja prekladačov.

1 Úvod do problematiky

V roku 2012 bol zverejnený prelomový výskum o časových útokoch na počítačové systémy [4]. Neskôr, v roku 2015, sa do úvah výskumníkov pridáva aj pamäť cache ako prostriedok na útoky cez bočné kanály [5]. V januári 2018 prichádza odhalenie zraniteľností Spectre, Meltdown a ich rôzne varianty [3, 6, 1], ktoré sa týkali takmer každého moderného procesora. Po zverejnení podrobností ohľadom týchto nových zraniteľností začali práce na opravách procesorov, jadier a OS. Výrobcovia CPU vydali niekoľko mikrokódových aktualizácií, vývojári Linuxové jadra taktiež prišli s množstvom záplat, ktoré avšak priniesli so sebou značný úpadok výkonnosti jadra. Toto zníženie výkonu sa novšie verzie jadra pokúšajú zmierniť a pomaly vrátiť výkon jadra na úroveň pred Spectre a Meltdown. Paralelne s tvorbou záplat začal výskum v oblasti obrany proti týmto zraniteľnostiam aj na strane prekladačov. Postupom času tvorcovia prekladačov prišli s novými opatreniami, ktoré vďaka možnostiam prekladača a jeho rôznych analýz, umožňujú aplikovať efektívnu obranu proti týmto zraniteľnostiam bez znateľného dopadu na celkový výkon.

2 Spectre, Meltdown a ich varianty

Moderné procesory používajú predikciu skokov a špekulatívne vyhodnocovanie na dosiahnutie maximálneho výkonu. Odhalenia v januári 2018 avšak priniesli

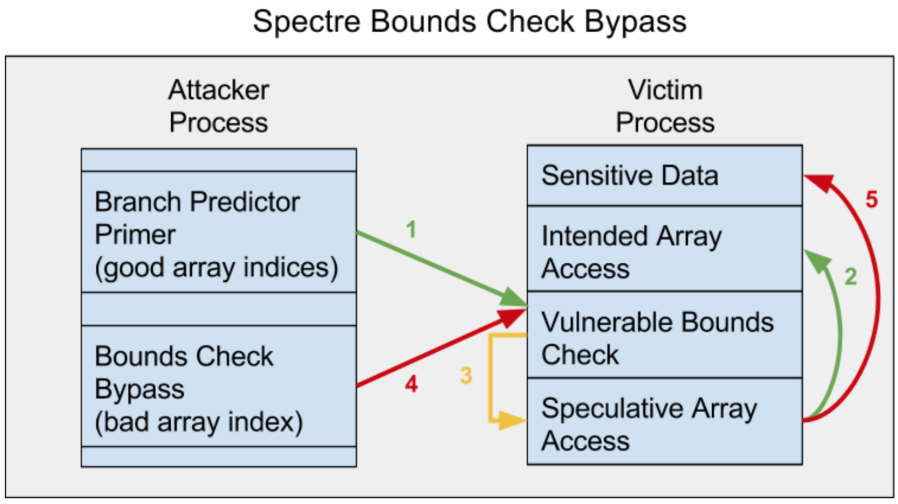
iný pohľad na tieto optimalizačné mechanizmy. Predikcia skokov a špekulatívne vyhodnocovanie stoja za najväčšími bezpečnostnými zraniteľnosťami poslednej doby. Cieľom tejto kapitoly je ozrejmiť princíp jednotlivých variánt, priebeh potenciálneho útoku a možnosť obrany proti nim.

2.1 Spectre varianta 1 - Bounds Check Bypass

Varianta 1 prináša novú triedu zraniteľností, o ktorej vývojári softvéru vôbec neuvažovali. Pre lepšie pochopenie tejto zraniteľnosti, uvažujme nasledujúci kód:

```
if (untrusted_index < array1_length) {
        unsigned char value = array1[untrusted_index];
        unsigned char value2 = array2[value * 64];
}
```

Kód obsahuje kontrolu hraníc pola pre uistenie sa, že `untrusted_index` je menší ako dĺžka pola `array1`, a teda slúži na zabránenie čítania pamäte za koncom poľa. Na prvý pohľad je toto korektné riešenie problému, no vôbec neberie v ohľad správanie moderných CPU zvané špekulatívne vyhodnocovanie. V takomto prípade CPU nemusí správne odhadnúť výsledok podmieneného skoku a môže vykonať špekulatívne čítanie poľa s `untrusted_index`, aj keď je hodnotovo väčší alebo rovný ako dĺžka poľa. Dochádza následne k čítaniu za hranice poľa. Takto získaná hodnota ďalej slúži ako index do poľa `array2`. Špekulatívne čítania môžu mať viditeľné bočné efekty v pamätiach cache, kde sa odhalia dáta, ktoré boli prečítané za hranicou poľa. CPU po zistení, že došlo k nesprávnej predikcií podmieneného skoku, výsledok špekulatívneho vyhodnocovania zahadzuje. Problémom avšak je, že už nezahodí vzniknuté zmeny v pamätiach cache a vzniká tak bočný kanál.

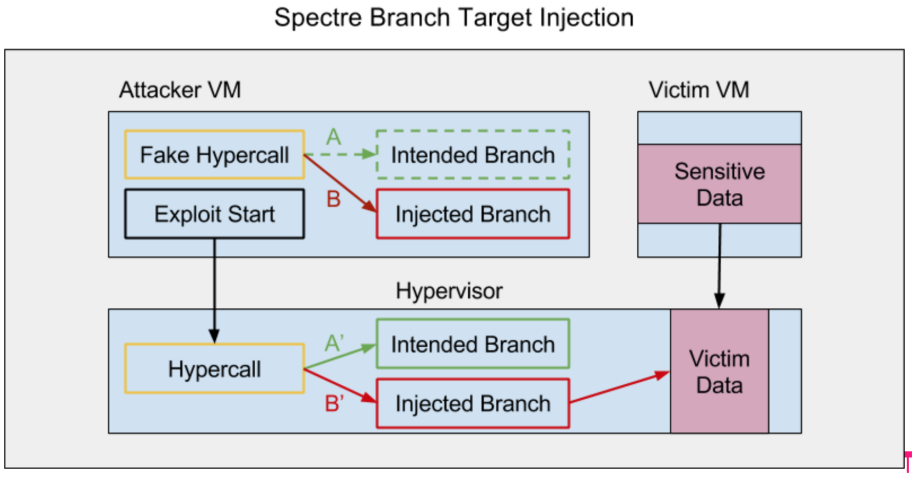


Obr. 1: Útočník opakovanie poskytuje správny vstup (1. šípka). Nedochádza len k čítaniu poľa (2. šípka), ale aj k ovplyvneniu prediktoru skokov a jeho budúcich rozhodnutí (3. šípka). Následne útočník použije nesprávny vstup (4. šípka), no vďaka natrénovanému prediktoru skokov, ktorý v tomto prípade predikuje, že sa bude skákať tam, kde sa skákalo aj v prípade správnych vstupov, získa prístup k citlivým dátam. (5. šípka).

Keďže tento vzor kódu, ktorého sa Spectre variant 1 týka, nie je až taký častý, dopad na výkon po aplikácií opatrení proti tejto variante je skôr zanedbateľný.

2.2 Spectre varianta 2 - Branch Target Injection

Táto varianta Spectre zneužíva špekulatívne vyhodnocovanie spôsobené rozhodnutiami prediktora nepriamych skokov. Nepriamy skok je inštrukcia, ktorá môže skočiť na viac, ako len na jedno miesto - príkladom môžu byť tabuľky funkcií alebo virtuálne metódy. Výskum ukázal, že útočník môže využiť tieto skoky, aby sa dostal na miesto, ktoré ho zaujíma.



Obr. 2: Kód útočníka pomocou falošných hypervolaní natrénuje prediktor nepriamych skokov aby preferoval len jeho vloženú vetvu (šípka B). Následne spustí pravé hypervolanie, ktoré programovo ma prejsť na cielenú vetvu (šípka A'), no namiesto toho dôjde k skoku na vloženú vetvu (šípka B'), čo umožní získanie prístupu k dátam obeti pomocou útoku cez bočný kanál.

V súčasnosti sú možné dve opatrenia proti tomuto útoku. Prvou je sada mikrokódových aktualizácií od výrobcov CPU, ktorá umožní programom spravovať stav prediktora skokov. Druhou možnosťou je zabránenie tomu, aby nepriame skoky bol ovplyvnené prediktorom skokov.

Implementované riešenie v prekladači Clang a GCC je známe ako *return trampoline* alebo *retpoline* [1]. Principiálne, namiesto použitia skokovej inštrukcie pre nepriame skoky, dochádza k uloženiu cieľovej adresy skoku na zásobník a zavolaniu *return* inštrukcie pre návrat. Rozdiel spočíva v tom, že návratové skoky používajú špeciálnu cache prediktora - tzv. *Return Stack Buffer (RSB)*, ktorej stav môže byť ľahko modifikovaný pred návratom, čím sa zruší možnosť prevedenia úspešného útoku.

Vo väčšine prípadov zabezpečený kód beží bez prediktora nepriamych skokov, čo znamená, že procesor musí čakať, kým správna adresa cieľa skoku nie je prečítaná, a až následne potom môže pokračovať vo vyhodnocovaní. Ak je cieľ skoku v hlavnej pamäti, môže sa jednať až o 100 nanosekúnd prerušeného vyhodnocovania.

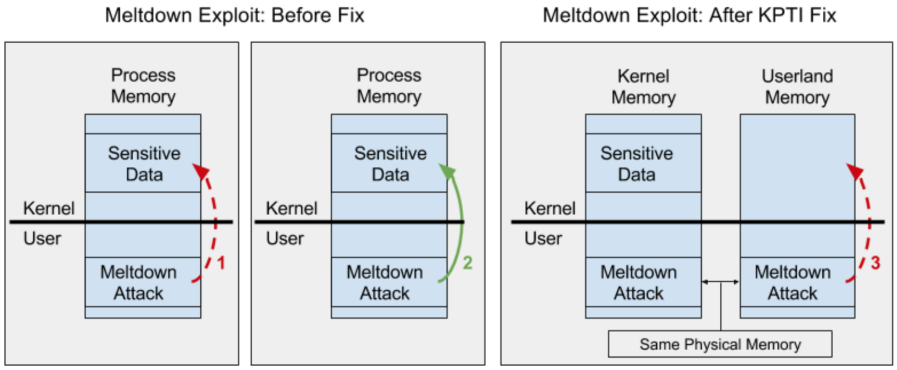
[1] `https://software.intel.com/security-software-guidance/api-app/sites/default/files/Retpoline-A-Branch-Target-Injection-Mitigation.pdf`

2.3 Meltdown

Meltdown porušuje základný predpoklad bezpečnosti operačného systému: aplikácia spustená v užívateľskom priestore nemá prístup k pamäti jadra. Toto obmedzenie je veľmi dôležité aby fungovalo, pretože pamäť jadra môže obsahovať citlivé informácie z inej aplikácie, napríklad heslá alebo kľúče. Na vynútenie tohto obmedzenia prístupu používajú operačné systémy tabuľky stránok na rozdelenie virtuálnej pamäte na dve časti - jedna pre jadro a druhá pre programy používateľa. Jadro je závislé na procesore, aby umožnil privilegovanejšiemu jadru pristupovať k obom častiam, pričom obmedzuje programy používateľa na časť určenú pre používateľa. Ukázalo sa však, že niektoré procesory toto obmedzenie nedodržiavajú.

Meltdown demonštruje problematiku vyhodnocovanie mimo poradia (tzv. *Out Of Order (OOO)*). Tento mechanizmus vyhodnocovania spôsobí, že procesor prečíta pamäť jadra bez overenia akýchkoľvek prístupových oprávnení. Môže tak dôjsť k vyzradeniu pamäti jadra do používateľského režimu na dostatočne dlhú dobu, aby takto odhalené dáta boli zachytené útokom z bočného kanála.

Opravou na strane Linuxu je zavedenie tzv. *kernel page-table isolation (KPTI)*.



Obr. 3: Pred Meltdownom sa predpokladalo, že kód z používateľského režimu nemôže pristúpiť k citlivý dátam v jadre (1. šípka). Ukázalo sa však, že to pravdou nie je (šípka 2). KPPT zamedzuje, aby boli dáta jadra zahrnuté v tabuľkách stránok používaných používateľským režimom, čím sa znemožní prevedenie Meltdown útoku (3. šípka).

3 Obrana proti Spectre v prekladači GCC

GCC od verzie 8 ponúka hneď niekoľko opatrení, akými je možné bojovať proti týmto novým zraniteľnostiam. Prekladač ponúka nasledovné možnosti:

`-mindirect-branch=choice`: Prevedie nepriame volania a skoky na *choice*. V predvolenom nastavení je táto možnosť nastavená na *keep*, ktorá ponecháva nepriame volania a skoky nezmenené. Varianta *thunk* prevádza nepriame volania a skoky na volania a *return thunk* miesta (tzv. medziskok, trampolína). Varianta *thunk-inline* prevádza nepriame volania a skoky na vnorené (*inlined*) volania a *return thunk* miesta. Varianta *thunk-extern* prevádza nepriame volania a skoky na externé volania a *return thunk* miesta v inom objektovom súbore. Vývojár môže toto chovanie nastaviť špecificky pre každú funkciu pomocou atribútu `indirect_branch`.

`-mfunction-return=choice`: Prevedie návraty z funkcií na *choice*. V predvolenom nastavení je táto možnosť nastavená na *keep*, ktorá ponecháva návraty z funkcií nezmenené. Varianta *thunk* prevádza návraty z funkcií na volania a *return thunk* miesta. Varianta *thunk-inline* prevádza návraty z funkcií na vnorené (*inlined*) volania a *return thunk* miesta. Varianta *thunk-extern* prevádza návraty z funkcií na externé volania a *return thunk* miesta v inom objektovom súbore. Vývojár môže toto chovanie nastaviť špecificky pre každú funkciu pomocou atribútu `function_return`.

`-mindirect-branch-register`: Vynucuje nepriame volania a skoky cez registre.

Okrem týchto možností ponúka vývojárom aj vstavanú funkciu na vloženie inštrukcie `LFENCE` na citlivé miesta v programe.

4 Obrana proti Spectre v prekladači Clang

Prekladač Clang okrem podpory naivného opatrenia založeného na vkladaní inštrukcie `LFENCE` prichádza s novým opatrením pre Spectre variantu 1 zvaným `Speculative Load Hardening` [2]. Povoliť toto opatrenie je možné od Clang 7.0 pomocou prepínača `-mspeculative-load-hardening`. Vývojár môže toto opatrenie povoliť pre určitú funkciu pomocou atribútu `speculative_load_hardening`.

Princíp tejto techniky spočíva v kontrole načítaní pomocou bezskokového kódu pre zaistenie, že tok programu nasleduje platný tok riadenia programu.

Uvažujme nasledovný kód v jazyku C pre ozrejmenie tejto techniky:

```
void leak(int data);
void example(int* pointer1, int* pointer2) {
        if (condition) {
                // ...
                leak(*pointer1);
        } else {
                // ...
                leak(*pointer2);
        }
}
```

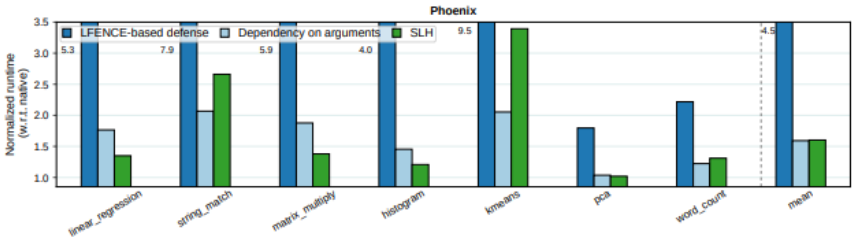
Po aplikovanej tejto techniky dôjde k transformácii na približne nasledovný kód:

```
uintptr_t all_ones_mask = std::numerical_limits<uintptr_t>::max();
uintptr_t all_zeros_mask = 0;
void leak(int data);
void example(int* pointer1, int* pointer2) {
uintptr_t predicate_state = all_ones_mask;
if (condition) {
// Assuming ?: is implemented using branchless logic...
predicate_state = !condition ? all_zeros_mask : predicate_state;
// ... lots of code ...
//
// Harden the pointer so it can't be loaded
pointer1 &= predicate_state;
leak(*pointer1);
} else {
predicate_state = condition ? all_zeros_mask : predicate_state;
// ... more code ...
//
// Alternative: Harden the loaded value
int value2 = *pointer2 & predicate_state;
leak(value2);
}
}
```

Uvažujme, že podmienka `condition` bola nesprávne predikovaná. Vidíme, že existuje dátová závislosť medzi podmienkou a nulovaním všetky ukazovateľov pred možným čítaním cez ne alebo medzi podmienkou a nulovaním všetkých načítaných bitov. Aj v prípade že sa vykoná špekulatívne vyhodnotenie, nie je možné že by došlo k odhalení tajných dát z pamäti. Tento prístup vyžaduje, aby koncový hardvér implementoval bezskokovú a nepredikovateľnú podmienenú aktualizáciu hodnôt v registroch. Všetky moderné architektúry túto vlastnosť majú, keďže je nevyhnutá pre správnu implementáciu kryptografických primitív v konštantnom čase.

Benchmark	`lfence`	**Load Hardening**	**Mitigated Speedup**
Google microbenchmark suite	-74.8%	-36.4%	**2.5x**
Large server QPS (using ThinLTO & PGO)	-62%	-29%	**1.8x**

Obr. 4: Porovnanie opatrení na Google microbenchmarkoch a ich vplyv na výkon



Obr. 5: Porovnanie opatrení na Phoenix benchmark suite a ich vplyv na výkon

Ako vidíme na obrázkoch 4 a 5, `LFENCE` riešenie na reálne programy má rapídny dopad na výkon. Z tohto dôvodu je pochopiteľný odpor a neochota akceptovať značné zníženie výkonu za cenu lepšieho zabezpečenia. Na druhej strane, `Speculative Load Hardening` prináša značne lepšie výsledky a prezentuje sa ako to najlepšie riešenie, aké zatiaľ existuje. V súčasne dobe v (nielen) LLVM komunite stojacej za prekladačom Clang prebiehaju ďalšie diskusie a aktívny výskum, ako toto riešenie zlepšiť.

Clang v boji proti Spectre variante 2 implementuje navrhované opatrenie *retpoline*, podobne ako aj GCC či ICC.

5 Bezpečnostné opatrenia v ICC

Spoločnosť Intel taktiež vyvíja svoj vlastný C/C++/Fortran prekladač, ktorý umožňuje generovať veľmi efektívny kód. Prekladač ponúka prepracované optimalizácie, podobne ako GCC či Clang. Významnou črtou prekladača je autovektorizácia kódu, ktorá prináša významný nárast výkonu koncových programom. Okrem optimalizácií avšak Intel pristúpil aj na implementáciu nových opatrení proti Spectre variantám, ktorým sa bude venovať práve tato kapitola.

5.1 Obrana proti Spectre variante 1

Prekladače od Intelu pri detekcii problematického vzoru (kontrola hranice poľa a načítanie z pamäti) automaticky vkladajú inštrukciu `LFENCE`. Okrem toho, Intel ponúka aj vstavanú funkciu `_mm_lfence()`, ktorá sa v rámci prekladača pretransformuje práve na inštrukciu `LFENCE`. Výhodou je, že vývojár môže použiť túto vstavanú funkciu a cielene ju vložiť na problematické miesta v zdrojovom kóde kde vie, že sa jedná o citlivé dáta. Tento krok môže zmenšiť pokles výkonu, ktorú by inak nastala po automatickom zabezpečení všetkých potenciálnych zraniteľných miest prekladačom.

5.2 Obrana proti Spectre variante 2

Opatrenie *retpoline* bolo úspešne naimplementované do C/C++/Fortran prekladačov od Intelu a ovláda sa pomocou prepínača `-mindirect-branch=choice` (analógia s tým, čo ponúka GCC prekladač).

6 Opatrenia proti Spectre a Meltdown útokom v MSVC

MSVC je C/C++ prekladač vyvíjaný spoločnosťou Microsoft pre OS Windows. Po odhalení zraniteľností ako sú Spectre a Meltdown, ktorý majú dopad na mnoho operačných systémov a moderných procesorov, sa aj Microsoft rozhodol implementovať do svojho prekladača sadu opatrení na lepšie zabezpečenie generovaného kódu proti zneužitiu týchto zraniteľností. Vývojárom sa odporúča stiahnutie najnovšej verzie MSVC prekladača v prípade, že ich vyvíjaný kód pracuje s citlivými dátami. Následne je potrebné skompilovať kód s prepínačom `/Qspectre` a nasadiť túto verziu zákazníkom čo najskôr. Podpora `/Qspectre` v MSVC prekladači je zahrnutá v Visual Studio 2017 version 15.6 Preview 4 a novšom.

6.1 Dopad na výkon

Testy vykonané MSVC tímom ukazujú, že dopad použitia `/Qspectre` je zanedbateľný [7]. Tím vyskúšal zostaviť celý OS Windows s `/Qspectre` a nezaznamenal regresie vo výkone. Zlepšenie výkonu vďaka špekulatívnu vyhodnocovaniu po aplikácii `/Qspectre` bolo síce stratené, no ako sa ukázalo, počet miest v zdrojových kódoch, kde bolo potrebné toto opatrenie aplikovať, je relatívne malý. Napriek tomu sa odporúča overiť dopad `/Qspectre` u každého produktu, u ktorého sa `/Qspectre` plánuje použiť. Ak vývojár o istej časti kódu vie, že je citlivá na výkon, je možné v tejto časti toto opatrenie vypnúť pomocou `__declspec(spectre(nomitigation))`.

6.2 Opatrenia proti Spectre v1

V rámci opatrenia proti Spectre v1 sú nevyhnutné softvérové zmeny na všetky aktuálne dotknutých CPU. Riešením je použitie inštrukcií, ktorý slúžia ako bariéra pre špekulatívne vyhodnocovanie. Pre procesory od Intel/AMD je odporúčanou inštrukciou `LFENCE`. ARM odporúča podmienený presun alebo podmienený výber inštrukcie, na novších architektúrach inštrukciu `CSDB`. Tieto inštrukcie zaistia, že špekulatívne vyhodnocovanie nedosiahne nebezpečný tok v programe za danou bariérou.

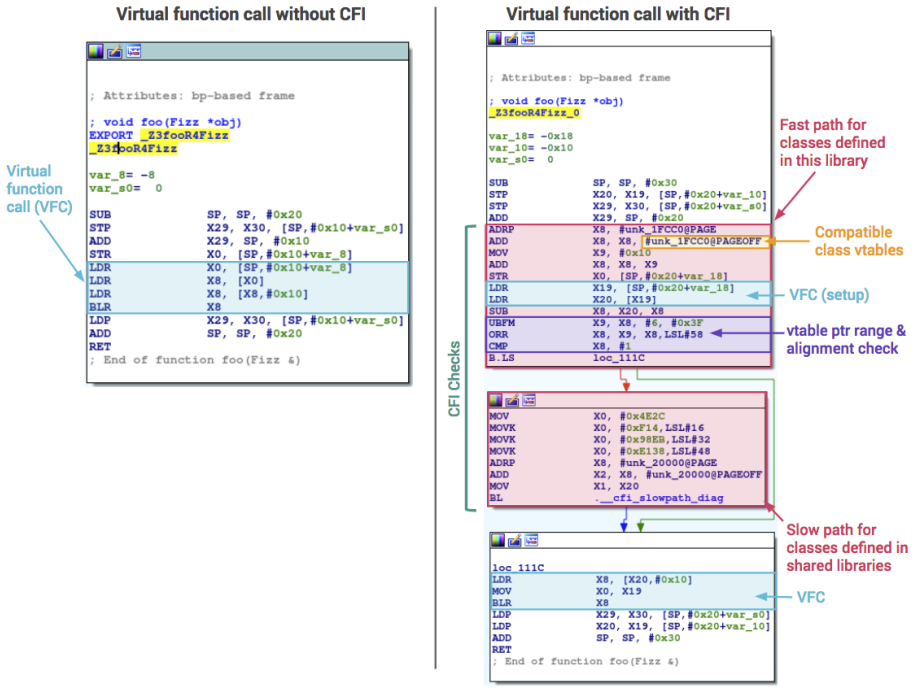
MSVC u prepínaču `/Qspectre` podporuje automatickú detekciu inštancií Spectre v1. Ak prekladač narazí na zraniteľnú časť kódu, automaticky vloží bariéru proti špekulatívnemu vyhodnocovaniu. Zároveň je nutné dodať, že implementovaná analýza inštancií Spectre v1 má taktiež svoje limity v rámci MSVC prekladača, a preto nie je možné garantovať, že budú zabezpečené všetky inštancie Spectre v1 pomocou `/Qspectre`.

7 Opatrenia v OS Android

Prechod na Clang ako na predvolený platformový prekladač v systéme Android 7.0 priniesol nové možnosti pre zvýšenie zabezpečenia systému. V prechádzajúcich verziách systému boli pridané opatrenia, ktoré sa zameriavajú na to, aby zneužitie chýb neviedlo k zraniteľnostiam, ktoré by mohli následne cieľom nových exploitov. V novej verzii systému Android 9.0 dochádza k rozšíreniu tejto sady opatrení. Pri detegovaní nedefinovaného chovania pomocou inštrumentácie inštrukcií za behu programu dôjde k jeho bezpečnému ukončeniu. Keďže sa softvér násilne ukončí, nie je možné vykonať zneužite chýb, ktoré nedefinované chovanie so sebou prináša.

7.1 Kontrola integrity toku riadenia programu

Dôležitou súčasťou moderných exploitov je fáza získania kontroly nad tokom riadenia programu útočníkom, a to napríklad poškodením ukazovateľov na funkcie alebo poškodením návratových adries. Po ovládnutí programu môže útočník spustiť vlastný alebo už existujúci kód programu na dosiahnutie svojho zámeru. Kontrola integrity toku riadenia programu (Control Flow Integrity, CFI) popisuje sadu opatrení, ktoré obmedzujú tok riadenia programu na graf volaní platných cieľov, ktoré boli vytvorené počas doby kompilácie [8, 9].



Obr. 6: Kontrola integrity toku riadenia programu v praxi

Obrázok 6 ukazuje funkciu, ktorá volaná virtuálna funkciu nad objektom, na úrovni strojového kódu s a bez použitia kontroly integrity toku riadenia programu. Bez tohto opatrenia dochádza k načítaniu ukazovateľa na tabuľku virtuálnych metód daného objektu a volaniu funkcie na očakávanom posune. Po aplikovaní tohto opatrenia, sa najskôr vykoná kontrola, či ukazovateľ patrí do povoleného rozsahu adries kompatibilných tabuliek virtuálnych metód. V prípade zlyhania tejto kontroly dochádza ku kontrole platných tried, ktoré sú definované v iných zdieľaných knižniciach. Ak ukazovateľ na tabuľku virtuálnych metód je neplatný cieľ, program sa ukončí. S obmedzením toku riadenia programu na malú množinu platných cieľov sa stáva zneužitie zraniteľností týkajúcich sa poškodenia pamäte čoraz náročnejšou alebo až nemožnou úlohou pre útočníka.

Implementácia týchto opatrení sa zameria na zabránenie modifikácie toku riadenia cez nepriame skoky, ako sú napríklad ukazovatele na funkcie alebo virtuálne funkcie. Platné ciele skokov sú definované ako vstupné body funkcií pre funkcie s očakávanou funkčnou signatúrou, čo umožňuje drasticky znížiť množinu cieľov, ktoré môže útočník využiť. Pre detekciu porušenia staticky vytvorenej množiny povolených cieľov je potrebné inštrumentovať nepriame skoky za behu programu. Po zistení porušenia (skok na neočakávaný cieľ) sa program bezpečne ukončí.

7.2 Detekcia pretečenia integeru

Už v systéme Android 7.0 boli predstavené opatrenia na detekciu pretečenia integeru v rámci mechanizmu na zistenie nedefinovaného chovania. Zameriavali sa zabezpečenie knižníc pracujúcich s multimédiami [10], ktoré boli častým cieľom rôznych útokov.

Toto nové opatrenie spôsobí bezpečné ukončenie programu akonáhle detekčný mechanizmus zistí pretečenie integeru. Samotná detekcia pretečenia je založená na inštrumentácií aritmetických inštrukcií, ktoré môžu viesť k pretečeniu za behu programu. Umožňuje zabrániť veľkému množstvu zraniteľností, ktoré súvisia s poškodením pamäti alebo odhalením informácií, kde ich hlavná príčina vzniku je práve pretečenie integeru. Prioritne bola táto detekcia aplikovaná na knižnice, ktoré spracovávajú komplexný neoverený vstup, alebo u ktorých boli ohlásené zraniteľnosti týkajúce sa pretečenia integeru.

7.3 Diskusia a záver

V rámci tohto článku boli predstavené najznámejšie zraniteľnosti posledného obdobia a možnosti obrany proti nim. Konkrétne boli prezentované opatrenia, ktoré boli implementované v prekladačoch na obranu proti týmto zraniteľnostiam. Aktuálne implementované riešenia sú na uspokojivej úrovni a za momentálne najlepšie bezpečnostné opatrenie proti Spectre je možné považovať `Speculative Load Hardening` v prekladači Clang. Na druhej strane, ani s týmto opatrením nie je možné zanedbať pokles výkonu programov, a preto štandardne

preklad programov neprebieha s prepínačmi prinášajúcimi tieto opatrenia na zvýšenie bezpečnosti programov a to najmä kvôli obavám súvisiacich s poklesom výkonu. Prekladače avšak majú dostatočné informácie o generovaných inštrukciách, a preto by bolo zrejme možné implementovať analýzu, ktorá by vedela podať informácie o zmenách vo výkone po použití týchto nových opatrení s väzbou na konkrétnu časť kódu, ktorá je problematická. Je možné, že vývojári by vďaka týmto novým informáciám mohli problematický kód upraviť tak, aby zníženie výkonu nebolo tak citeľné, prípadne by mohli tento kód úplne refaktorovať a vyhnúť sa potenciálne nebezpečným prístupovým vzorom do pamäte. Ďalším zaujímavým faktorom je aj príchod umelej inteligencie do oblasti prekladačov, ktorá prinesie masívne vylepšenia nie len z pohľadu lepších optimalizácií a generovania kódu, ale aj v oblasti statickej a dynamickej analýzy programov. Lepšie a kvalitnejšie analýzy sú základom pre bezpečnú budúcnosť IT a je dôležité smerovať investície aj do výskumu a vývoja nových prostriedkov a opatrení, ktoré priamo síce negenerujú zisky spoločnosťiam, no bez nich môžu raz tieto spoločnosti prísť o oveľa väčšie finančné prostriedky po úspešnom útoku na ich zraniteľné miesta, ktoré neboli zabezpečené.

Literatúra

[1] Spectre Attacks: Exploiting Speculative Execution
Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin
https://spectreattack.com/spectre.pdf, 3.1. 2018.

[2] Speculative Load Hardening
Chandler Carruth
https://docs.google.com/document/d/1wwcfv3UV9ZnZVcGiGuoITT_61e_Ko3TmoCS3uXLcJR0,
23. 3. 2018

[3] Reading privileged memory with a side-channel
Jann Horn, Project Zero
https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html, 3. 1. 2018.

[4] Compiler mitigations for time attacks on modern x86 processors
Jeroen V. Cleemput, Bart Coppens, Bjorn De Sutter
https://dl.acm.org/citation.cfm?id=2086702, 4. 1. 2012

[5] Robust and Efficient Elimination of Cache and Timing Side Channels
Benjamin A. Braun, Suman Jana a Dan Boneh
https://arxiv.org/pdf/1506.00189.pdf,

[6] Meltdown and Spectre: Exploits and Mitigation Strategies
Chris Stevens, Nicolas Poggi, Thomas Desrosiers a Reynold Xin
https://databricks.com/blog/2018/01/16/meltdown-and-spectre-exploits-and-mitigation-strategies.html, 16. 1. 2018.

[7] Spectre mitigations in MSVC
MSVC Compiler team
https://devblogs.microsoft.com/cppblog/spectre-mitigations-in-msvc/, 15. 1. 2018.

[8] Compiler-based security mitigations in Android P
Ivan Lozano, Information Security Engineer
https://android-developers.googleblog.com/2018/06/compiler-based-security-mitigations-in.html, 17. 6. 2018.

[9] Diving into Control Flow Integrity
Hanno Böck
https://blog.fuzzing-project.org/57-Diving-into-Control-Flow-Integrity.html, 31. 3. 2017.

[10] Hardening the media stack
Dan Austin, Jeff Vander Stoep, Android Security team.
https://android-developers.googleblog.com/2016/05/hardening-media-stack.html, 5. 5. 2016.