MENEMBUS CONTROL-FLOW ENFORCEMENT TECHNOLOGY (CET) DAN BRANCH TARGET IDENTIFICATION (BTI) DENGAN FUNCTION-ORIENTED PROGRAMMING (FOP) (Studi kasus: Linux kernel)

Suryo Bramasto [1)], Muhammad Ramli [1)]
1) Program Studi Teknik Informatika Institut Teknologi Indonesia
E-mail: suryo.bramasto@iti.ac.id

Abstrak

Perlindungan pada CPU mutakhir telah memperkenalkan berbagai rintangan. ARM telah memiliki mekanisme "Pointer Authentication" dan "Branch Target Identification" untuk menangani otentikasi alamat memori dan pointer, dan kemudian Intel telah menindaklanjutinya dengan mekanisme Shadow Stack dan Indirect Branch Targeting, yang juga dikenal sebagai Control-Flow Enforcement Technology. Perlindungan ini membuat hampir mustahil untuk menggunakan metode code reuse biasa seperti return-oriented programming (ROP) dan jump oriented programming (JOP). Penelitian ini menunjukkan pendekatan baru untuk menggunakan Function-Oriented Programming (FOP) sebagai teknik untuk digunakan dalam lingkungan tersebut. Demonstrasi FOP dalam kernel Linux menunjukkan kemampuan FOP untuk unggul dalam situasi dunia nyata yang kompleks.

Kata kunci: *function-oriented programming, gadgets, kernel, linux, proteksi.*

Pendahuluan

Pada *Return-Oriented Programming* (ROP), penyerang menguasai tumpukan panggilan untuk membajak alur kendali program, lalu mengeksekusi rangkaian instruksi mesin yang telah dipilih dengan cermat, dan sudah ada di memori mesin, yang disebut *gadget* [1]. Setiap *gadget* biasanya diakhiri dengan instruksi pengembalian dan terletak di dalam subrutin dalam kode program dan/atau pustaka yang sudah ada. Jika dirangkai bersama, *gadget-gadget* ini memungkinkan penyerang untuk melakukan operasi acak pada mesin yang menerapkan pertahanan yang dapat menggagalkan serangan. *Return-Oriented Programming* (ROP) telah ada selama lebih dari 15 tahun [2]. Sejak saat itu, teknik penggunaan ulang kode lainnya telah terbukti berhasil dalam keadaan tertentu seperti *Jump Oriented Programming* (JOP) [3]. Seiring berkembangnya kedua teknik ini dalam dekade terakhir, perlindungan baru telah diciptakan untuk membatasi kegunaannya. Beberapa di antaranya termasuk *Control-Flow Enforcement Technology* (CET) [4] untuk prosesor Intel dan *Pointer Authentication* (PAC) serta *Branch Target Identification* (BTI) [5] untuk prosesor ARM. Tujuan dari perlindungan ini adalah untuk membatasi penggunaan serangan penggunaan ulang kode seperti ROP dan JOP dalam suatu program.

BTI melibatkan penerapan langsung tujuan pendaratan spesifik untuk cabang tidak langsung, baik berupa instruksi lompatan/panggilan register maupun memori. Biasanya ditempatkan di awal fungsi yang rentan terhadap referensi tidak langsung, direktif *landing pad* ini berfungsi sebagai pengaman. Pada prosesor Intel, instruksi *landing pad* ini adalah `ENDBR64`. Jika lompatan atau panggilan tidak langsung gagal mendarat pada instruksi yang ditentukan, sebuah *trap* akan dipicu. Langkah perlindungan ini bertujuan untuk mengurangi efektivitas serangan JOP. *Functional Oriented Programming* (FOP) melibatkan penggunaan seluruh fungsi sebagai *gadget*. Berbeda dengan konsep "*gadget*" dalam skenario ROP atau JOP, yang biasanya hanya terdiri atas beberapa instruksi, seperti "POP RDI; RET;" klasik, FOP memperluas *gadget* untuk mencakup seluruh fungsi.

Tujuan pemanfaatan seluruh fungsi sebagai *gadget* adalah untuk memanfaatkan dua kemampuan dari fungsi. Pertama, kemampuan untuk memanfaatkan suatu fungsi sebagaimana mestinya. Contohnya, dengan mengendalikan dua parameter pemanggilan fungsi, seperti strcpy, manipulasi nilai dalam memori menjadi mungkin; dan dapat diperluas ke fungsi apa pun yang menyertakan instruksi *landing pad* awal di Glibc, seperti `mprotect` atau `syscall`. Kemampuan kedua diperoleh melalui efek samping yang muncul dari pemanggilan suatu fungsi. Karena register parameter biasanya dianggap volatil, memulihkan atau melindunginya tidak ada gunanya. Hal ini mengakibatkan nilai register parameter berpotensi berguna setelah pemanggilan fungsi.

Kemampuan yang dijelaskan bergantung pada *gadget* dispatcher yang andal. Dalam contoh yang disebutkan sebelumnya, *dispatcher* tidak boleh mengubah register-register dari argumen yang sangat rentan berubah di antara pemanggilan *gadget* FOP. Makalah ini mengkaji *dispatcher* semacam itu yang biasanya ditemukan dalam Glibc, dan dipanggil dalam eksekusi normal. Karena pemanggilan fungsi *dispatcher* tidak memerlukan parameter, register parameter tidak diatur ulang di antara pemanggilan. Hal ini memungkinkan *gadget* FOP berikutnya untuk menggunakan parameter register yang ditetapkan oleh pemanggilan sebelumnya. Dengan menggunakan kerentanan kerusakan memori dan *dispatcher* ini, rantai FOP dapat dibangun untuk mendapatkan eksekusi.

Studi Pustaka

Function-Oriented Programming (FOP) adalah evolusi dari kelas serangan *code-reuse* yang sebelumnya dikenal lewat ROP (*Return-Oriented Programming*) dan JOP (*Jump-Oriented Programming*). Ketika mitigasi seperti DEP/NX dan ASLR mencegah eksekusi kode yang disuntikkan serta berbagai mekanisme lain memperkecil peluang *gadget-level chaining,* FOP muncul sebagai pendekatan yang bekerja pada tingkat fungsi penuh: bukannya merangkai potongan instruksi kecil (*gadget*s), FOP menyusun "program" jahat dengan memanggil fungsi-fungsi yang sudah ada di *binary* atau *library* target. Karena menggunakan blok yang lebih besar dan mengikuti ABI/prolog-epilog fungsi yang wajar, urutan panggilan fungsi yang dimanipulasi bisa tampak lebih natural dan sulit dideteksi oleh teknik yang hanya mencari pola *gadget*.

Secara konseptual, proses FOP melibatkan identifikasi fungsi-fungsi yang berguna, misalnya fungsi I/O, alokasi memori, atau fungsi yang mengubah *state,* yang bila dikombinasikan dapat mencapai tujuan seperti mengambil data sensitif atau memicu tindakan tertentu. Berbeda dengan ROP/JOP yang fokus pada instruksi individu, FOP membutuhkan cara untuk memaksa program memanggil fungsi-fungsi tersebut dalam urutan dan dengan argumen yang dikontrol, seringkali melalui korupsi struktur data atau kontrol alur yang memengaruhi bagaimana dan kapan fungsi dipanggil. Karena level abstraksinya lebih tinggi, FOP dapat melewati beberapa deteksi yang dirancang khusus untuk *gadget-level reuse* dan menimbulkan tantangan baru bagi sistem mitigasi yang hanya memeriksa target loncatan atau pola instruksi.

Secara formal, definisi kelas serangan FOP, terkategori menjadi kemampuan fungsi yang bermanfaat bagi penyerang dan menunjukkan skenario teoretis atau simulasi di mana chaining fungsi dapat memenuhi tujuan eksploitasi. Telah terdapat analisis tentang bagaimana mitigasi modern seperti *Control-Flow Integrity* (CFI), shadow stacks, dan proteksi mikroarsitektur mempengaruhi kemungkinan FOP, serta titik-kelemahan yang masih ada. Selain itu, perlu evolusi alat deteksi dan teknik analisis otomatis. Detektor dan generator *exploit* generasi berikutnya perlu memahami semantik fungsi dan representasi menengah (misal p-code/IR) agar dapat mengevaluasi kombinasi fungsi yang mungkin disalahgunakan.

FOP memperluas permukaan serangan *code-reuse* karena banyak program memiliki fungsi-fungsi yang, bila disusun bersama, dapat mencapai efek kompleks tanpa *gadget-level chaining*. Mitigasi tradisional yang efektif melawan ROP/JOP tidak selalu cukup terhadap FOP; misalnya, CFI yang hanya membatasi target *control-flow* mungkin masih mengizinkan urutan pemanggilan fungsi yang sah namun disalahgunakan. Oleh karena itu, deteksi otomatis harus berevolusi untuk mempertimbangkan konteks pemanggilan fungsi dan semantik panggilan, bukan sekadar pola instruksi.

Dari perspektif defensif, memahami FOP penting untuk merancang proteksi yang lebih efektif. Pendekatan defensif yang diusulkan meliputi penerapan CFI yang lebih halus dan berbasis konteks (memverifikasi tidak hanya target pemanggilan tetapi juga konteks dan tipe argumen), monitoring perilaku semantik untuk mendeteksi urutan panggilan fungsi yang tidak biasa, pembatasan permukaan API sensitif melalui prinsip *least privilege* dan pemisahan hak istimewa, serta penggunaan *sandboxin*g dan teknik isolasi lain untuk membatasi dampak potensi penyalahgunaan fungsi. Selain itu, penggabungan analisis statis dan dinamis yang memahami

representasi menengah kode dapat membantu mengidentifikasi kombinasi fungsi yang memungkinkan tindakan berbahaya.

Metodologi Penelitian

Metodologi penelitian yang diterapkan pada artikel ini diawali dengan menetapkan definisi formal dan model ancaman FOP, guna merumuskan bagaimana serangan ini berbeda dari ROP/JOP dan menetapkan asumsi-asumsi lingkungan misal ketersediaan fungsi di *binary/library* dan kemampuan pengendalian data/struktur untuk memengaruhi pemanggilan fungsi). Selanjutnya dilakukan inventarisasi dan klasifikasi *building blocks* pada tingkat fungsi yakni mengidentifikasi kategori fungsi yang berguna bagi penyerang (I/O, alokasi/manipulasi memori, kontrol proses), menganalisis ABI/*calling-convention*, dan bagaimana argumen dapat dikontrol melalui korupsi data atau rekayasa struktur program.

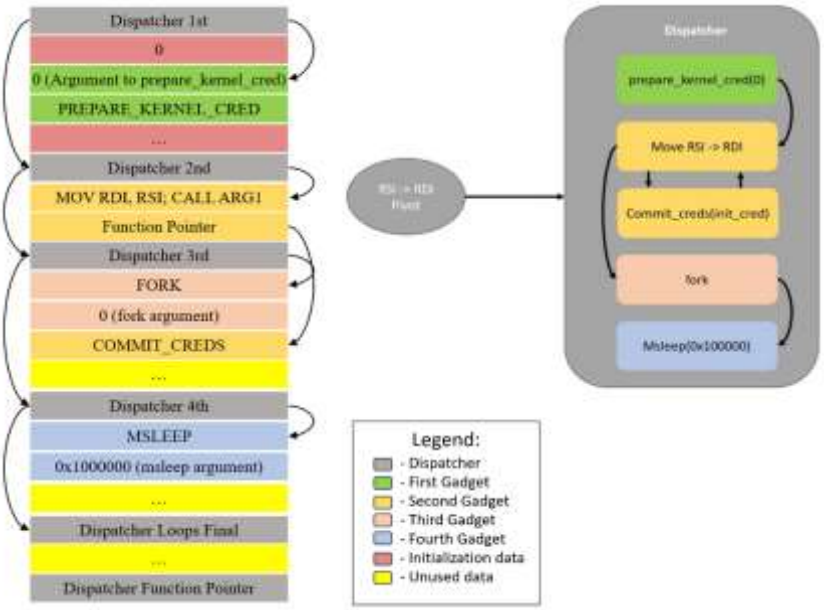
Guna menemukan kandidat fungsi dan mengevaluasi apakah fungsi-fungsi itu dapat "dirangkai", metodologi yang dipakai meliputi analisis statis (*disassembly*, pembuatan CFG, pencarian simbol/ekspor, analisis p-code/IR) dan analisis dinamis (instrumentasi *runtime* untuk mengamati perilaku fungsi dan efek sampingnya). *Automatic tooling* digunakan untuk *scanning library/binary* mencari fungsi yang memenuhi prasyarat tertentu. Selanjutnya disusun *proof-of-concept* atau skenario simulasi (dalam lingkungan terisolasi) yang menunjukkan bagaimana *chaining* fungsi dapat mencapai goal tertentu, misal membaca berkas sensitif atau memicu eksekusi proses, kemudian mengevaluasi efektivitas mitigasi modern (CFI, ASLR, NX/DEP, *shadow stacks*, SMEP/SMAP, dan lain-lain) terhadap teknik FOP.

Terakhir, disusun rangkuman implikasi defensif dengan merekomendasikan perbaikan seperti CFI yang lebih halus/berbasis-konteks, monitoring semantik urutan panggilan, dan pengurangan permukaan API, serta menunjukkan kebutuhan alat deteksi dan analisis otomatis yang mengerti semantik fungsi (bukan hanya pola *gadget* instruksi) agar mitigasi dapat berkembang menghadapi serangan bergranularitas-fungsi.

Hasil dan Pembahasan

Versi Linux kernel yang menjadi target pada penelitian ini adalah 5.19.17 (November 2022), dimana telah mendukung penuh implementasi CET, dukungan penuh terhadap IBT untuk landing pad, berikut mekanisme proteksi baru seperti fungsi prepare_kernel_cred. Mekanisme-mekanisme proteksi yang ada pada kernel versi "baru" (setelah 5.18) membatasi rantai serangan ROP secara signifikan dengan menerapkan mekanisme new check terhadap fungsi-fungsi yang sering menjadi target serangan. Pada setiap serangan di tingkat kernel, keberhasilan serangan sangat bergantung pada ketersediaan celah kerentanan yang memungkinkan utilisasi target-target potensial, dimana pada kernel versi setelah 5.18 telah terdapat juga mekanisme yang menutup celah-celah kerentanan.

Dampak FOP terhadap kernel merupakan salah satu poin utama penelitian ini. Kemampuan untuk mencapai serangan yang efektif menggunakan FOP *gadget*s, alih-alih ROP *gadget*s atau JOP *gadget*s, merupakan definisi implementasi FOP yang sukses. Penelitian ini mengkaji primitif FOP dalam target Libc untuk ruang pengguna, sehingga tidak terdapat perincian lengkap primitif untuk bagian kernel Linux. Karena fleksibilitas dan variasi primitif berkorelasi langsung dengan jumlah *gadget* yang ditemukan, basis kode yang lebih besar menyiratkan besarnya kapabilitas yang sama atau lebih besar dibandingkan dengan primitif Libc. Mengingat kernel Linux berisi *gadget* hampir sepuluh kali lebih banyak, penelitian ini menduga bahwa terdapat *gadget* yang cukup untuk mendemonstrasikan primitif yang sama seperti yang diidentifikasi sebelumnya. Satu aspek tambahan yang perlu diperhatikan adalah bahwa meskipun jumlah fungsi dan *gadget* potensial lebih besar, tidak ada identifikasi gawai yang memungkinkan kemampuan untuk memanfaatkan nilai yang disimpan dalam RAX. Keterbatasan ini terdapat baik di ruang pengguna maupun di ruang kernel untuk sistem Intel. Gambar 1 menunjukkan rantai FOP akhir yang digunakan dalam kernel Linux yang menghasilkan Linux *privilege escalation*.



Gambar 1. Rantai kernel FOP di dalam memori

Kerentanan ini memungkinkan kerusakan memori melalui luapan *heap*, dimana penyerang dapat mengendalikan struktur "msg_msg", yang mengakibatkan kebocoran kernel Linux dan alamat heap kernel sehingga dapat melewati KASLR. Hal ini kemudian memungkinkan penimpaan struktur "pipe_buffer" beserta "_ops pointer", yang mengakibatkan pemanggilan fungsi arbitrer dengan register RSI yang menunjuk ke memori yang dikendalikan. PoC awal menerapkan teknik ini untuk *pivot* tumpukan ke RSI dan memulai rantai ROP. Dalam contoh FOP ini, *pivot* ke *dispatcher* FOP justru terjadi. Masalah yang muncul adalah tidak adanya identifikasi *dispatcher* FOP yang berasal dari RSI selama pengujian. Untuk mengatasi masalah ini, diperlukan penggunaan perangkat *pivot* dari RSI ke RDI seperti yang ditunjukkan pada Gambar 2, yang memungkinkan penggunaan *dispatcher* yang ditunjukkan pada Gambar 1.

```
KERNEL 0xffffffff813aaf00:
  Results:
    ...
    RDI: 0xfffffffffffffc8 + [40 + RSI]
    ...
  Symbolic Target:
        [16 + [0x78 + [40 + RSI]]]
```

Gambar 2. Gadget dari RSI pivot ke RDI

Setelah *dispatcher* dimulai, proses normal eskalasi kernel pun dimulai. Rantai FOP yang ditunjukkan menggunakan fakta bahwa kontrol argumen pertama dalam panggilan ke "prepare_kernel_cred" dengan argumen bernilai nol, mengembalikan salinan struktur kredensial kernel. Tidak ada *gadget* yang memiliki kemampuan untuk menggunakan nilai dalam register RAX. Dengan kata lain, struktur kredensial kernel yang dibuat oleh prepare_kernel_cred yang dikembalikan dalam RAX tidak berguna. Namun, "prepare_kenel_cred" memiliki efek samping menyimpan struktur "init_cred" dalam register RSI setelah fungsi tersebut. Memindahkan *pointer* ke RDI dan memanggil "commit_cred" memungkinkan rantai FOP untuk menetapkan *privilage* saat ini ke root, sehingga menyelesaikan *privilage escalation*.

Kemampuan terakhir yang diciptakan adalah untuk berhasil kembali ke ruang pengguna sebagai *elevated user*. Pendekatan ROP untuk kembali ke ruang pengguna adalah dengan menyimpan status penyimpanan dalam tumpukan terkontrol dan kembali menggunakan fitur kernel yang dirancang untuk kembali ke ruang pengguna. Agar tetap valid dengan spesifikasi tumpukan bayangan yang ditetapkan sejauh ini, tidak ada modifikasi pada tumpukan yang dapat terjadi selama serangan,

sehingga pendekatan ini tidak layak. Ada dua pendekatan yang dapat dilakukan rantai FOP untuk kembali ke ruang pengguna. Yang pertama adalah merancang *dispatcher* di sepanjang rantai FOP dan keluar dengan aman setelah semua eskalasi hak istimewa selesai. Hal ini dimungkinkan karena kernel akan menganggap bahwa semuanya telah berjalan dengan benar karena tidak ada modifikasi yang terjadi pada tumpukan atau ruang memori. Tetapi karena perangkat *dispatcher* yang digunakan menggunakan R12 sebagai mekanisme penghitungan, maka menjadi ini tidak layak. Hal ini dapat ditentukan melalui pengujian dan dengan melihat dispatcher pada Gambar 1 dan mengidentifikasi bahwa register R12 diatur oleh RDI dan RSI. Dalam kasus ini, *dispatcher* RSI dan RDI mengarah ke memori yang dikendalikan, sehingga R12 menjadi angka substansial yang tidak realistis untuk diiterasi. Pendekatan kedua adalah memanfaatkan teknik *telefork*.

Teknik *telefork* beroperasi dengan memanggil *fork system call* dari dalam kernel. Selanjutnya, tindakan ini memunculkan *thread* kedua di dalam aplikasi ruang pengguna tepat di titik tempat eksploitasi awalnya berhenti. Sementara itu, *thread* asli tetap berada di dalam kernel Linux dan diatur untuk menunggu tanpa batas waktu melalui fungsi msleep. Pengaturan ini mengakibatkan kernel *thread* berhenti terkunci dan menciptakan kesan bahwa tidak ada yang berubah dari perspektif eksternal. Karena pendekatan ini hanyalah pemanggilan fungsi normal, pendekatan ini merupakan target utama untuk digunakan oleh FOP. Aspek terakhir ini memungkinkan serangan FOP untuk berhasil kembali ke ruang pengguna, sehingga menghasilkan Linux *privilage escalation* yang berhasil, yang dimungkinkan hanya dengan menggunakan gadget FOP.

Kesimpulan

Demonstrasi serangan FOP yang bekerja di dalam kernel Linux menunjukkan kemampuan FOP untuk bekerja di lingkungan dunia nyata yang kompleks, dan membantu memperkuat FOP sebagai teknik potensial yang ampuh untuk eksploitasi di masa mendatang dalam lingkungan modern yang membatasi ROP dan serangan *code reuse* lainnya. Kontribusi ini digabungkan untuk memenuhi pedoman penelitian ini dalam mendemonstrasikan kegunaan dan kapabilitas FOP.

Ucapan Terima kasih

Penelitian ini dibiayai oleh Institut Teknologi Indonesia Tahun Akademik 2025/2026.

Daftar Pustaka

[1] J. Ravichandran, W. T. Na, J. Lang, and M. Yan, “PACMAN,” in Proceedings of the 49th Annual International Symposium on Computer Architecture, New York, NY, USA: ACM, Jun. 2022, pp. 685–698. doi: 10.1145/3470496.3527429.

[2] S. Ognawala, F. Kilger, and A. Pretschner, “Compositional Fuzzing Aided by Targeted Symbolic Execution,” Oct. 2019.

[3] R. C. Goluch, “Trust, transforms, and control flow: A graph-theoretic method to verifying source and binary control flow equivalence,” Iowa State University, 2021. doi: 10.31274/etd-20210609-59.

[4] M. Lipp et al., “Meltdown,” Commun ACM, vol. 63, no. 6, pp. 46–56, May 2020, doi: 10.1145/3357033.

[5] S. Matsuoka, “Fugaku and A64FX: the First Exascale Supercomputer and its Innovative Arm CPU,” in 2021 Symposium on VLSI Circuits, IEEE, Jun. 2021, pp. 1–3. doi: 10.23919/VLSICircuits52068.2021.9492415.