Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Hálózati Rendszerek és Szolgáltatások Tanszék
CrySyS Adat- és Rendszerbiztonság Laboratórium

Pintér Olivér <oliver.pinter@gmail.com>

MEMÓRIAVÉDELMI MEGOLDÁSOK VIZSGÁLATA ÉS MEGVALÓSÍTÁSA FREEBSD OPERÁCIÓS RENDSZERBEN

KONZULENS

Dr. Bencsáth Boldizsár

BUDAPEST, 2013

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott **Pintér Olivér**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2013. 05. 17.

...……

Pintér Olivér

Összefoglaló

Napjainkban az informatika és informatikai szolgáltatások elterjedtek. Az élet minden területére beszivárogott, és az emberek napjainak szerves részét képezi. Reggel felkelnek, előveszik az okostelefonjukat és közösségi oldalakat néznek, a buszon híreket olvasnak, este éttermet vagy szórakozóhelyeket keresnek az interneten. Ezt az információéhséget és terhelést ki kell szolgálni.

A legnagyobb rendelkezésre állással rendelkező tartalomszolgáltatók általában valamilyen szerver operációs rendszert használnak. A Netcraft[46] felmérései alapján ezek túlnyomó többségében FreeBSD, Microsoft Windows Server 2008 és Linux rendszerek.

A munkám során FreeBSD rendszerekkel foglalkozom, több okból kifolyólag. A FreeBSD a Linuxokhoz és Windowsokhoz képest kevésbé elterjedt rendszer, kisebb fejlesztői bázissal rendelkezik, azonban bizonyos területeken így is versenyképes velük. Teljesítményben sok helyen meg is előzi őket. A FreeBSD egy nyílt forrású rendszer, mely fejlesztése során első szempont a teljesítmény volt. A korlátozott fejlesztői kapacitások miatt így más területek háttérbe szorultak. Az egyik ilyen terület a biztonság. A rendszer tartalmaz biztonsági megoldásokat, amelyek magasabb szinten befolyásolják a rendszer működését. Ezek a teljesség igénye nélkül: jails, DAC, MAC, BSM és BSM-re épülő megoldások. Fejlesztésük a TrustedBSD projekt keretén belül készült el, melyet részben a DARPA támogatott.

A rendszer azonban a mai napig szinte mentes az alacsony szintű proactive védelmi megoldásoktól. A dolgozatomban ezek állapotának felmérésével foglalkoztam és egy (kettő) lehetséges megoldás kiválasztása után azt implementáltam. A fontossága miatt választásom az ASLR-re és újdonsága miatt az SMAP-ra esett.

Abstract

Today, information technology and information services are prevalent. They’re present in all aspects of life and form an integral part of people's days. People get up in the morning and take their smart phones, browsing social networking sites and reading news on the bus, and at night they looking for restaurants or night clubs online. This hunger for information and load must to be served.

The content providers that have the greatest availability generally have a server operating system in use. According to Netcraft's surveys, the overwhelming majority of them are, FreeBSD, Microsoft Windows Server 2008 and Linux systems.

The topic of my thesis deals with the FreeBSD system for many reasons. FreeBSD is less commonly used than Linux or Windows, and has a smaller developer base; however, in some areas it is still competitive with them, and is leader in performance. FreeBSD is an open-source system, where the priority in development was performance. Due to the limited capacity of development in other areas are overshadowed. One such area was security. The system includes security solutions that affect the operation of the system at a higher level. These are without attempting to be comprehensive: jails, DAC, MAC, BSM and BSM solutions built on. Their development was completed within the TrustedBSD project – which was partially supported by DARPA.

However, the FreeBSD system still does not have a lot of proactive security solutions. My thesis deals with the assessment of their condition and after choosing one (two) possible solution that has been implemented. The choice fell on ASLR and SMAP.

1 Bevezetés

1.1 Feladat értelmezése

A feladat, megvizsgálni, hogy a jelenlegi FreeBSD 10-CURRENT (a legfrissebb fejlesztői változat, 2013. február – 2013. május) milyen védelmi megoldásokat tartalmaz a lehetséges megoldások közül. Ezt követően egy nem implementált megoldást implementálok, és megvizsgálom valós körülmények között, kitérve a teljesítményre is, mivel a FreeBSD projekt számára az elsők között van a teljesítmény is.

1.2 Tervezés célja

A FreeBSD operációs rendszer biztonságosabbá tétele lehetőleg optimális módon, minimális sebességvesztés mellett.

Az elkészült kódok követik a FreeBSD szellemiségét, így törekszem az olvashatóságra és a követhetőségre. A tervezés során első ízben már meglévő (tipikusan PaX Team által tervezett) algoritmusok rendszerbe illesztésére törekszem, ehhez elmélyült FreeBSD ismeretek szükségesek.

1.3 Feladat indokoltsága

Mint ahogy az a dolgozatból kiderül, a FreeBSD operációs rendszer még 2013-ban is mentes sok alapvető védelmi mechanizmustól, amit más konkurens rendszerek tartalmaznak (Microsoft Windows XP SP3 és újabb Windowsok, különböző Linux disztribúciók).

A FreeBSD operációs rendszert sok helyen használják, ISP-k, tartalomszolgáltatók, hálózati infrastruktúra készítők és IT biztonsági cégek is. Emellett tervek fogalmazódtak meg bennem a FreeBSD operációs rendszerrel. A közeljövőben vagy FreeBSD commiter (olyan személy, aki rendelkezik írásjoggal a hivatalos FreeBSD verziókövető rendszeréhez) szeretnék lenni, vagy a FreeBSD operációs rendszer forkolásával kívánok létrehozni egy biztonságos és friss BSD variánst, mely annyira követi a szülő operációs rendszert, amennyire csak lehet.

1.4 Szakdolgozat felépítése

- A mű „Támadási formák” részében rövid áttekintést adok a rendszerek sérülékenységeit kihasználó technikákról és támadási formákról.
- A „Memóriavédelmi megoldások” részben számításba veszem az ellenük felhasználható védelmi megoldásokat.
- A „FreeBSD” részben a FreeBSD operációs rendszert érintő részegységeiről adok egy áttekintést.
- A „Intel x86-64” részben az Intel x86-64 processzor pár tulajdonságáról fogok részletesebben írni, melyek kapcsolódnak a témámhoz.
- Az ezt követő részekben („SMAP implementálása FreeBSD kernelben” és „ASLR megvalósítása FreeBSD-ben”) pedig a konkrét tervezési és megvalósítási megoldásokat írom le, melyek építenek az előző fejezetekre.
- Végezetül az „Értékelés” részben értékelem és összegzem a megoldásokat valamint a belőlük leszűrt tapasztalatokat és eredményeket.

2 State-of-the-art feltárás

2.1 Proactive security

A klasszikus biztonsági megoldások és megvalósítások egy már bekövetkezett támadás és/vagy káreset felkutatásával és utólagos vagy folyamatlagos kezelésével foglalkoznak, ezt hívjuk általánosabban **reactive security**-nek is. Ilyen megoldások a ma is forgalomban kapható vírusirtók, tűzfalak és forensics eszközök, mivel többségük egy, már ismert minta vagy szignatúra alapján dolgozik. Természetesen ezek az eszközök is egyre nagyobb tudással lesznek felvértezve és mozdulnak el proactive irányba.

Ezzel szemben a **proactive security** megoldások a hangsúlyt egy esetleg támadás megelőzősére fektetik. A készítői és alkalmazói megpróbálnak mindig egy lépéssel a támadó előtt lenni és megakadályozni egy esetleges betörés sikerességét vagy legalább nagyságrendekkel megnehezíteni azt. Ilyen megoldások a '90-es évek végétől kezdtek terjedni, leginkább a **PaX** és az **OpenWall** kezdeményezéseként. Az általuk kifejlesztett technológiákat később nagyobb software cégek is átvették, többek között a Microsoft is a Windows rendszerekhez.

2.2 Támadási formák

Ebben a szakaszban az x86-64 rendszereket leginkább érintő támadási formákról adok egy rövid áttekintő leírást. Bizonyos támadások más architektúrán nem működhetnek.

2.2.1 NULL pointer dereferencia

Mint ahogy a neve is mutatja, ezzel a támadási formával olyan pointereket használnak, melyek értéke NULL, azaz a 0 memória címre mutat. Ez a támadási forma régebben volt elterjedt. Manapság többféle megoldással is védekeznek ellene. Unix rendszerek esetében a run-time loader a program betöltése után a programhoz tartozó első lapot (mely a 0-ás virtuális memória címtől kezdődik) unmappeli a program címteréből, így ha valamilyen okból kifolyólag egy program dereferálna egy olyan pointert, ami NULL-t tartalmaz, akkor az operációs rendszer a programot terminálja.

Régebben speciális esetben ezek után is lehetett használni ezt a megoldást. Az unmappeléstől függetlenül, ha az *mmap(...)* függvény segítségével újra felmappelték a címtérbe a 0-ás címtől kezdődő memória régiót és ide helyezték a shellcode-ot vagy a címet, amely a shellcode-ra mutatott.

A kernel számos függvénypointert tartalmaz, melyek többsége futás közben van inicializálva. Bizonyos konstellációk esetén ezek a pointerek *NULL* értéken maradnak, és ebben az esetben van lehetőség a támadás kihasználására.

A következő kódrészlet bemutatja az egyszerűsített kernel oldali hibát:

```
void (*kf_ptr)(void) = NULL;

int
foo_read_on_device_hanlder()
{
     [...]
     /* ellenorzes nelkul, igy az erteke NULL */
     kf_ptr();

     [...]
}
```

Amikor a kernel végrehajtja a *read(...)* hívást a *char dev*-en, akkor lefut a *kf_ptr(...)* függvény, azonban a *kf_ptr NULL*-ra mutat, így az aktuális process címterének *NULL* címén található utasításait hajtja végre a kernel.

User space oldalról pedig a következő kódrészlet:

```
char shellcode[] = “SHELLCODE helye”;

void
bar(void)
{
     int dev=0;
     char dummy[4096];

     mmap(NULL, PAGE_SIZE, PROT_READ|PROT_WRITE|PROT_EXEC,
          MAP_FIXED|MAP_PRIVATE|MAP_ANON, -1, 0);
     memcpy(NULL, shellcode, sizeof(shellcode));

     dev = open(“/dev/echo”, O_RDWR);
     read(dev, dummy, PAGE_SIZE);
}
```

Itt felmappeljük a *NULL* címtől kezdődő lapot, és elhelyezzük rajta a shellcode-ot, ezt követően meghívjuk rajta a sérülékeny *read(...)* függvényt.

Erre is született egy korlátozó megoldás, melynek lényege, hogy *mmap(...)*-elni, csak egy bizonyos virtuális memória cím felett enged a rendszer, ez tipikusan egy lap méretű vagy ennek kettő hatvány szorosa. x86-64 rendszeren a lapméret 4096 byte, a minimális VM cím FreeBSD esetében pedig 4 kbyte, ha a security.bsd.map_at_zero sysctl értéke 0.

Védekezés ellene: VM kezdőcím megemelése, UDEREF, SMEP és SMAP.

2.2.2 Buffer overflow

2.2.2.1 stack overflow

Stack overflow vagy magyar nevén verem túlcsordulás tipikusan két esetben fordul elő. Az egyik, ha nagyszámú rekurzív függvényhívás történik; a másik, ha nagyméretű lokális változó van foglalva (nagy tömb a stacken).

A futás végeredménye programhiba, az operációs rendszer terminálja a folyamatot.

Stack overflow-t kernel space-ben könnyebb elérni, mivel a kernel stackek tipikusan nagyságrendekkel kisebbek, mint a user space programok stack-jei. FreeBSD esetében a kernel stack 2 lap, azaz 8192 byte x86-64 esetében, ezzel szemben a user stack több mega esetleg giga byte is lehet.

2.2.2.2 stack buffer overflow (stack smashing)

A probléma oka általában a hiányzó vagy téves határ ellenőrzés vagy integer over vagy underflow. Egy sérülékeny kód a következő:

```
#include <string.h>

void foo (char *bar)
{
   char  buff[12];

   strcpy(buff, bar);
}

int main (int argc, char **argv)
{
   foo(argv[1]);
}
```

Ha programot első paraméterként 12 karakternél hosszabb string-gel hívjuk meg, akkor a *foo(...)* függvényben található buff tömböt elkezdjük írni, és mivel a *strcpy(...)* függvény nem ellenőriz hosszt, ezért az írás nem fejeződik be a tömb "végénél", hanem a stack-en levő szenzitív adatokat is felülírjuk Egy megfelelően előkészített string esetén a program flow-t kontrollálni tudjuk.

Védekezés ellene ASLR-rel, NX stack-kel vagy SSP-vel történik.

2.2.2.3 heap overflow

A heap overflow[27] ugyancsak a buffer overflow-k családjába tartozik. A stack overflow-nál komplexebb támadást igényel.

A heap-en a memóriát dinamikusan allokáljuk. Ezen allokációk nyilvántartása általában egy segéd struktúrával történik, melyek magukban foglalják a memória régiókat is. A támadás célja ezeknek a struktúráknak a felülírása és a control flow eltérítése.

Védekezés ellene ASLR-rel és NX bittel történik.

2.2.3 Format string támadások

A format string támadások[32][33] programozói hibára és / vagy hanyagságra vezethetők vissza. A printf függvénycsalád szignatúrája igényel egy string-ként megadott paramétert, mely leírja, hogy az azt követő paraméterek milyen módon legyenek kiírva.

Ha a paraméter char tömb típusú (erős közelítéssel alias string) és ezt a buffer-t közvetlenül vagy közvetetten tudjuk befolyásolni, akkor tetszőleges értéket kiolvashatunk a segítségével.

Egy egyszerű példa, mely format string támadáshoz vezethet:

```
...
char buff[256];
scanf("%s", buff);
printf(buff);
...
```

A baj pedig a *printf(...)* függvénycsalád természetes működéséből fakad. A formátum string-ként megadott %-ot követő paramétereknek megfelelő értékeket veszi le a stack-ről. Ha inputként egy crafted inputot adunk, akkor annak megfelelően fogja végigpörgetni a stacket. Ez manapság általános esetben infoleak-hez jó. Például, ha a *"%x %x %x %x"* format string-et adjuk be, akkor a stack-ről az első 4 integert olvashatjuk ki.

Régebben vagy speciális fordítási direktívát használva, a *printf(...)* függvénnyel írni is lehet, nem csak olvasni a stack-ről. Ehhez *%n* formátumot kell alkalmazni, és paraméterként a *printf(...)* stack-jén a célcímet kell tartalmazni, ahova a *%n*-ig kiírt byte-ok száma kerül.

```
...
char *s="foo";
int  size;
printf("%s%n", s, &size);
...
```

A kód lefutása után a *size* változó értéke 3 lesz, mivel az *s* pointer egy 3 hosszú string-re mutat és a *"foo"* 3 karakterből áll.

Ha ez kombináljuk a lefelejtett format string-gel és mi tudjuk megadni, hogy mit, hol, hova és milyen módon írjon ki a *printf(...)*, akkor szinte bármit módosítani tudunk a memóriában.

Védekezés ellene ASLR-rel és NX bittel történik.

2.2.4 Return-oriented Programming (ROP)

A ROP[28][29] az újabb támadási formák közé tartozik. Érdekessége, hogy kódinjektálás nélkül vagyunk képesek a control flow-t eltéríteni. A támadás képes kijátszani az NX-et, mivel futtatható kód nem kerül a célterületre.

A támadás itt is egy buffer overflow-val kezdődik, mellyel a return address-t felülírjuk, ezt követően az előzőleg a buffer-be injektált speciális shellcode-ra kerül a vezérlés. Ez a shellcode stack frame-ekből áll, mely stack frame-ek a program és az általa használt függvénykönyvtárak kódjait használja fel – így más, már memóriában levő futtatható kódra hivatkozunk és emiatt nem kell futtatható kódot injektálni. Ezeket az apró építőköveket – melyek a futó program részei – gadget-eknek hívjuk. Ezek a gadget-ek *ret* utasításra kell végződniük, innen is kapta a nevét a támadás.

Védekezés ellene ASLR-rel történik.

2.2.5 Jump-oriented Programming (JOP)

A JOP[30] a ROP-hoz hasonló támadási forma, a különbség annyi, hogy itt a ret helyett indirekt jmp és call utasításokat tartalmazó gadget-eteket keressük. A ROP elleni egyik védekezés hívta életre, mivel a ret utasítások eliminálhatók a programból, ugrások és memória műveletek helyettesítésével. Az ugró utasítások ezzel szemben a programok szerves részét képezik, és eliminálásuk nehezebb.

Védekezés ellene ASLR-rel történik.

2.3 Memóriavédelmi megoldások

2.3.1 Memória írhatóságát és futtathatóságát korlátozó megoldások

2.3.1.1 PaX MPROTECT

A PaX MPROTECT[40] restriction megoldás lényege, hogy a *mmap(…)* és *mprotect(…)* interfész korlátozásával segíti megakadályozni új futtatható kód megjelenését a process címterében.

FreeBSD esetén a *vm_entry_t* struktúrában található *vm_prot_t protection* és *vm_prot_t maxprotection* elemekre kellene alkalmazni a megszorításokat, ha a FreeBSD tartalmazna PaX MPROTECT-et.

A megszorítások a következőkre terjednek ki:

- megakadályozza a futtatható anonymous mapping létrehozását,
- megakadályozza az írható-futtatható file mapping létrehozását,
- megakadályozza a csak olvasható-futtatható file mapping-ek olvashatóvá is tételét, kivéve, ha relokációt végzünk ET_DYN ELF file-on (non-PIC megosztott könyvtár),
- megakadályozza a nem futtatható mapping futtathatóvá tételét.

Ez a védelem sok előny mellett hátrányokkal is járhat, mivel az alkalmazás készítők hanyagságból vagy lustaságból nem megfelelően használják az *mmap(…)* és *mprotect(…)* interface-t. Ez leginkább a futás közben generált kódokra igaz, melyeket utána végre akarunk hajtani (JIT, just in time framework-ök).

A FreeBSD **nem** tartalmaz PaX MPROTECT biztonsági megoldást.

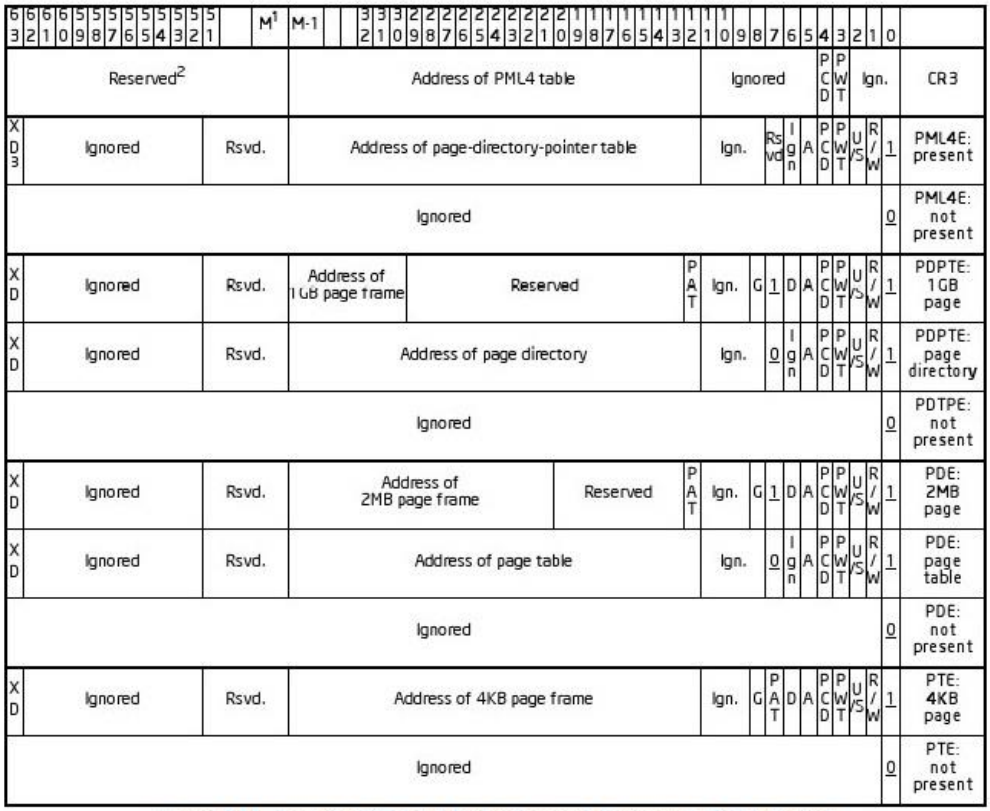
2.3.1.2 W^X

A W^X[42] az OpenBSD software-s megoldása az NX-re – mely a PaX PAGEEXEC[50]-ből merít ötleteket – , olyan esetekben, amikor a hardware nem tartalmaz hozzá kiegészítést. A megoldás a különböző ELF szekciókat szétszedi, és a védelmi szintnek megfelelően újrarendezi, melyek védett memória címeken helyezkednek el. Ez a módszer x86 esetében érvényes.

A FreeBSD **nem** tartalmaz W^X biztonsági megoldást.

2.3.1.3 XD

Az XD (XD bit – eXecute Disable) az Intel fejlesztése, mely segítségével lapokat tudunk nem futtathatónak jelölni. A szolgáltatás rendelkezésre áll az XD kiterjesztést tartalmazó 32 bites processzorokban és 64-bites x86-64 processzorokban.



63	62:M	M-1:32	31:12	11:9	8	7	6	5	4	3	2	1	0	
Reserved[2]			Address of PML4 table	Ignored					PCD	PWT	Ign.			CR3
XD	Ignored		Rsvd.	Address of page-directory-pointer table	Ign.		Rsvd.	Ign.	A	PCD	PWT	US	R/W	1	PML4E: present
Ignored														0	PML4E: not present
XD	Ignored		Rsvd.	Address of 1 GB page frame	Reserved	PAT	Ign.	G	1	D	A	PCD	PWT	US	R/W	1	PDPTE: 1GB page
XD	Ignored		Rsvd.	Address of page directory			Ign.		0	Ign.	A	PCD	PWT	US	R/W	1	PDPTE: page directory
Ignored														0	PDTPE: not present
XD	Ignored		Rsvd.	Address of 2MB page frame	Reserved	PAT	Ign.	G	1	D	A	PCD	PWT	US	R/W	1	PDE: 2MB page
XD	Ignored		Rsvd.	Address of page table			Ign.		0	Ign.	A	PCD	PWT	US	R/W	1	PDE: page table
Ignored														0	PDE: not present
XD	Ignored		Rsvd.	Address of 4KB page frame			Ign.	G	PAT	D	A	PCD	PWT	US	R/W	1	PTE: 4KB page
Ignored														0	PTE: not present

Figure 4-11. Formats of CR3 and Paging-Structure Entries with IA-32e Paging

NOTES:
1. M is an abbreviation for MAXPHYADDR.
2. Reserved fields must be 0.
3. If IA32_EFER.NXE = 0 and the P flag of a paging-structure entry is 1, the XD flag (bit 63) is reserved.

Figure 1: XD bitek előfordulása[1]

Az NX meglétét a *CPUID.80000001H:EDX.NX[bit20]* alapján tudjuk ellenőrizni[9].

Lényegében ez a hardware oldali megvalósítása a W^X-nek bizonyos korlátozásokkal. Önmagában nem jelent megoldást minden problémára, de elősegíti más technológiákkal együtt a hatékony védekezést.

FreeBSD alatt használt védelmi megoldás.

2.3.2 Stack canary (SSP)

A védelem lényege, hogy a stack frame-en belül az adatokat tartalmazó rész és a controlt tartalmazó (legfőképp a visszatérési címet) rész közé byte sorozatokat szúrunk be, melyeket később ellenőrzünk. Stack Smash Protection – SSP[41] – néven is szoktunk rá hivatkozni. A megoldás – mint ahogy a nevéből következik – csak a stack-et védi, így a head overflow ellen nem véd.

FreeBSD alatt használt védelmi megoldás.

A stack canary-nak 3 fő típusa van: terminator canary, random canary, random XOR canary.

2.3.2.1 Terminator canary

Ez a legegyszerűbb és a legkönnyebben kijátszható canary típus. Arra épít, hogy a legtöbb buffer overflow-t kihasználó támadás valamilyen string műveletre alapoz. Következményképpen a canary-t *NULL*-okból, *CR*-ekből, *LF*-ekből és *-1*-ből állítják össze.

Megkerülése azért egyszerű, mert a támadó ismerheti a canary értékét, és azt el tudja helyezni az exploitban vagy shell-kódban.

2.3.2.2 Random canary

Ez a típus már biztonságosabb, a lényege, hogy a programon belül egy védett globális változóban található egy random generált érték, melyet a program indításakor hoz létre az operációs rendszer. Ezt követően minden stack frame létrehozásakor és törlésekor ellenőrzésre kerül ez az érték.

2.3.2.3 Random XOR canary

Hasonló a Random canary-hoz, illetve arra épít, azzal a kiegészítéssel, hogy a globális canary értéke össze van XOR-olva a control adatok egy részével, így egy adott függvényhívás esetén specifikus canary-t kapunk és nem egy globális egész programra kiterjedő canary-t.

2.3.3 ASLR

Az ASLR[10] – Address Space Layout Randomization – célja, hogy véletlenszerűséget vigyen a processzek címterébe, ez által megnehezítve az exploitok lefutását.

Egy program címtere több részből áll, mint például a *text* szegmens, a *data* szegmens, a *bss*, a *stack* és a program által használt megosztott könyvtárak. Ezen felül a címtérbe lehetnek mappelve file-ok és memória régiók sokasága az *mmap(...)* rendszerhívás segítségével.

Az ASLR ezen szekciók kezdőcímeit hivatott randomizálni. Ezen felül, a program belépési pontja is randomizálható. Itt kétféle futtatható file típust különböztetünk meg: *ET_EXEC* és *ET_DYN*. Az *ET_EXEC* esetén a program maga nincs felkészítve relokációra, a benne található szimbólumok abszolút elérésűek a program belépési pontjától számítva, így ebben az esetben a belépési pont randomizációja nem triviális. *ET_DYN* esetben már maga a bináris fel van készítve a relokációra, így ebben az esetben egyszerűbb dolgunk van. A binárisokat az *exec_loader* tipikusan *mmap(...)* függvényhívás segítségével mappeli be a címtérbe.

A FreeBSD **nem** tartalmaz ASLR biztonsági megoldást.

2.3.3.1 mmap randomizáció

Az mmap randomizáció – PaX Team által RANDMMAP[11] néven referált megoldás – az *mmap(...)* rendszerhívást hivatott randomizálni. A randomizációt a címen minden *mmap(...)* hívás esetén alkalmazzuk, kivéve, ha a *MAP_FIXED* flag-gel jelezzük a kernelnek, hogy a megadott címre szeretnénk kérni az adott *vm_address* régiót.

2.3.3.2 stack randomizáció

Stack randomizáció – RANDUSTACK[12] – esetén a stack kezdőcímét randomizáljuk. Itt kétféle megoldás van. Az egyszerűbb az úgynevezett *stack gap* használata, amikor az adott lapon belül eltoljuk a stacket N darab bittel. A bonyolultabb megoldás pedig az előbbi megoldást is felhasználva ezen felül még a program indulásakor véletlen címre mappeli a stack kezdetét.

Létezik egy RANDKSTACK[13] is, mely a kernel stacket hivatott randomizálni.

2.3.4 UDEREF

Az UDEREF[14][15] megoldás azt a célt tűzte ki maga elé, hogy biztosítsa a kernel space – user space határokat és hiba esetén a kernel ne dereferáljon user space memóriát, ha az explicit nincs engedélyezve. Ha nem engedélyezett dereferálás történik, akkor a kernel *Oops*-ot vagy kernel *panic(...)*-ot dob az adatok sikeres kiolvasása helyett.

Ez azért fontos, mert így a kernel exploitok egy nagy halmazát hatástalanítani lehet. Ez azért működhet, mivel előtte legitim módon shell kódot kell a kernelbe injektálni, de az UDEREF miatt nem lehet egyszerűen csak a user space memóriát cím szerint dereferálni.

Az UDEREF egy software-es megoldás, x86-on hatékonyan implementálható minimális sebességvesztéssel szegmentáció alkalmazásával. Az x86-64-es CPU azonban nem tartalmazza a szegmentációt, így kerülő megoldásokat kell alkalmazni.

Az UDEREF a user space - kernel space átmenetnél unmappeli az eredeti user space címteret és átmappeli egy másik címtérre, ami non-exec/supervisor jogokkal bír, így ha a kernel innen próbál kódot futtatni, akkor *Oops*-ot vagy *panic(...)*-ot kapunk.

A FreeBSD **nem** tartalmaz UDEREF biztonsági megoldást.

2.3.5 Segvguard

Az ASLR sérülékenységét hivatott kijavítani. Az ASLR bruteforce alkalmazásával kijátszható. A servguard[16][17] önmagában nem egy memóriavédelmi megoldás. A megoldás lényege, hogy minden a kernel által terminált program után feljegyezzük a terminálás idejét és okát. Ezen információkat aggregáljuk, és egy limitet állítunk be rá: az adott idő alatt tolerálható hiba mennyiségét. A limit átlépése után a kernel addig nem enged új folyamatot létrehozni, ameddig a limit le nem telt.

A FreeBSD **nem** tartalmaz Segvguard biztonsági megoldást.

2.3.6 KERNEXEC

A KERNEXEC[48][49] a W^X esetében említett PAGEEXEC és az MPROTECT kernel space beli megfelelője. Segít megakadályozni, hogy kernel space kód kernel spaceben user space kódot lefuttasson.

A FreeBSD **nem** tartalmaz KERNEXEC biztonsági megoldást

2.3.7 SMEP

Az SMEP[44] (Supervisor-mode execution Prevention) ugyancsak az Intel által kifejlesztett technológia, melynek célja, hogy a kernel space kód ne tudjon user space memórián elhelyezkedő kódot végrehajtani. Ez lényegében az előbb említett KERNEXEC hardware oldali megfelelője.

FreeBSD alatt használt védelmi megoldás.

2.4 FreeBSD

2.4.1 Felépítés

A FreeBSD kernel tervezésekor törekedtek az egyszerűségre és átláthatóságra. A kód alapjában véve két jól elkülönülő részre osztható[6]: gépfüggő (*machine-dependent, MD*) és gép független (*machine-independent, MI*) kódokra. Ennek a szeparációnak a lényege, hogy egy adott megoldást vagy driver-t csak egyszer kell megírni *MI* módon, és ez az *MI* módon megírt réteg nem tartalmaz semmilyen *MD* kódot, és az – alsóbb – architektúránként változó, de azonos interface-t nyújtó *MD* réteget használja.

Az *MI* rész tovább osztható, ezek a teljesség igénye nélkül a következők:

- **basic kernel facilities – alap kernel szolgáltatások**: timer és rendszer idő kezelés, descriptor kezelés és process kezelés
- **memory-management support – memória menedzsment**: lapozás és lapcsere
- **generic system interfaces – általános rendszer interface-ek**: I/O kontrol és párhuzamos műveletek descriptor-okon
- **file system – file rendszerek**: file-ok, mappák, elérési utak fordítása, file zárolás és I/O buffer kezelés
- **terminal-handling support – terminál kezelés**: pseudo terminál interface és terminál felügyelet
- **inter-process communication facilities – folyamatok közötti kommunikáció**: socket-ek
- **networking – hálózatkezelés**: kommunikációs protokollok és általános hálózati erőforrások

Ezek közül jelenleg a memóriakezelésre és a virtuális memóriára fókuszálok.

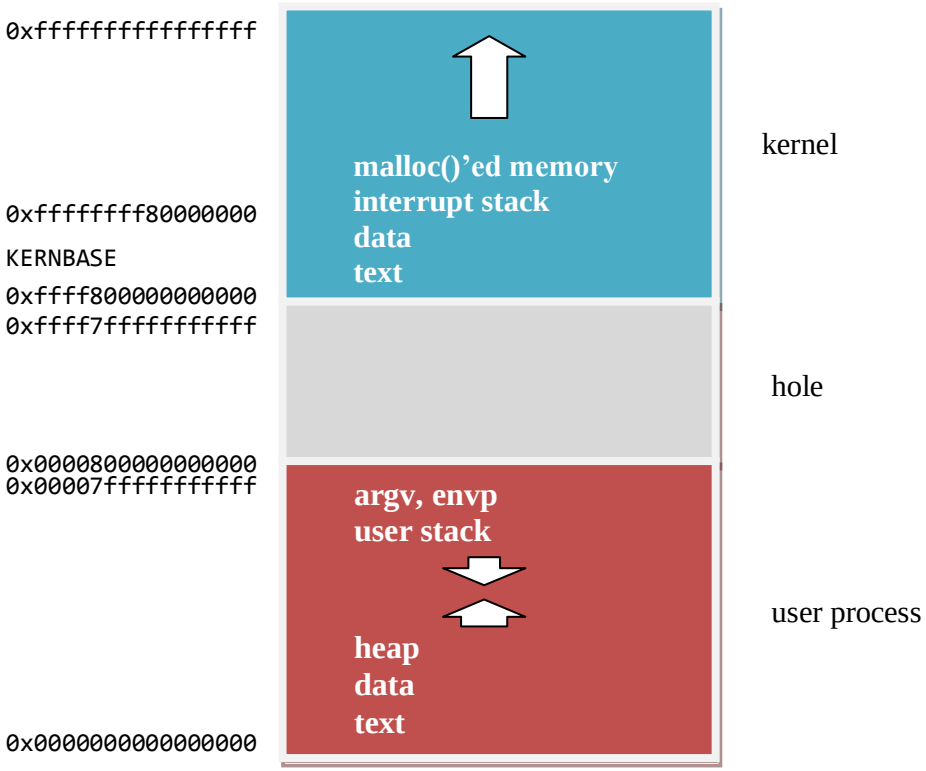
2.4.2 VM felépítés

2.4.2.1 Address space

A *sys/amd64/include/vmparams.h* file-ból:

```
/*
 * Virtual addresses of things.  Derived from the page directory and
 * page table indexes from pmap.h for precision.
 *
 * 0x0000000000000000 - 0x00007fffffffffff   user map
 * 0x0000800000000000 - 0xffff7fffffffffff   does not exist (hole)
 * 0xffff800000000000 - 0xffff804020100fff   recursive page table (512GB
slot)
 * 0xffff804020101000 - 0xffffdfffffffffff   unused
 * 0xffffe0000000000 - 0xfffffeffffffffff   1TB direct map
 * 0xffffff0000000000 - 0xffffff7fffffffff   unused
 * 0xffffff8000000000 - 0xffffffffffffffff   512GB kernel map
 *
 * Within the kernel map:
 *
 * 0xffffffff80000000                        KERNBASE
 */
```

Látható a következő ábrán, hogy a FreeBSD kernel – mint a legtöbb kernel – a negatív címtérben fut x86-64 architektúrán. A user space memória pedig a pozitív címtéren helyezkedik el.



FreeBSD address space amd64 architektúrán

Ezen belül is, a user space címtér érdekel minket első sorban, az ASLR relációjában.

2.4.2.2 A vmspace

A *vmspace* adatstruktúra írja le a FreeBSD rendszerben egy process címterét. Ez magában foglalja a virtuális memória állapotát, az *MI* és az *MD* réteget vonatkozólag is. Minden egyes *vm_map_entry* egy mappelt címet tartalmaz. Számunka ezek a legfontosabbak jelen helyzetben, mivel ezeket fogom módosítani.

A *vmspace* struktúra ki lesz egészítve két változóval, melyeket a későbbiekben mutatok be.

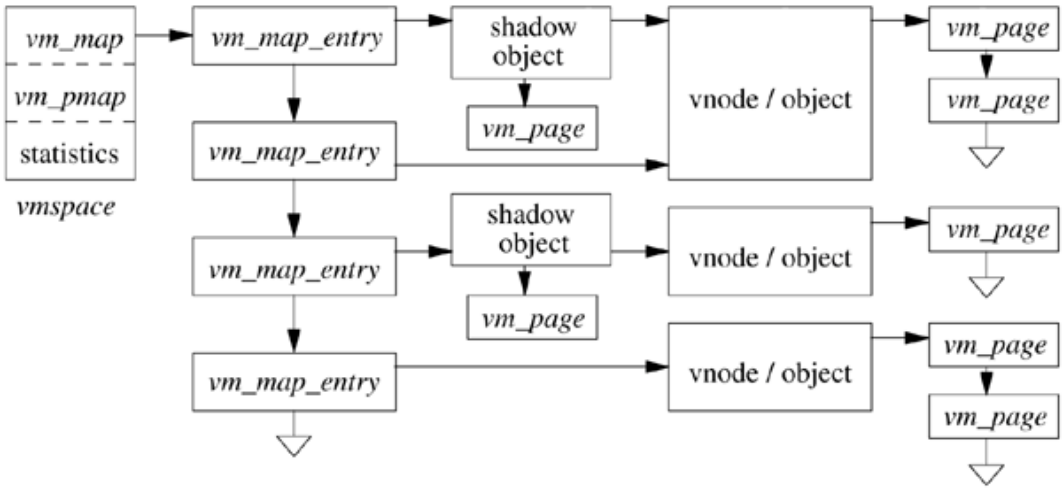


Figure 2: vmspace struktúra[24]

2.4.2.3 A stack

X86-64 esetében a stack lefelé terjeszkedik, ami annyit tesz, hogy kezdőcíme alapértelmezetten *0x00007fffffffffff*. Ahogy növekszik a stack úgy ez a cím csökken. A stacket használjuk *AUX* infók átadására is, melyeket a *run-time link-editor* (rtld) használ. Ugyancsak itt adjuk át a futtatandó programnak a paramétereit és a környezeti változókat is.

Position	Contents	Frame
8n+16(%rbp)	argument eightbyte n	Previous
	...	
16(%rbp)	argument eightbyte 0	
8(%rbp)	return address	Current
0(%rbp)	previous %rbp value	
-8(%rbp)	unspecified	
	...	
0(%rsp)	variable size	
-128(%rsp)	red zone	

Figure 3: Stack Frame Base Pointer-rel FreeBSD x86-64 alatt[26]

A stack-hez a *vmspace* struktúrában egy *vm_map_entry* tartozik, mely leírja a tulajdonságait.

A stack-en stack frame-ek találhatók, melyeket x86-64-en az *rbp* regiszter címez meg. Egy ilyen stack frame látható a Figure 3 ábrán.

2.4.2.4 Az mmap

Az *mmap(...)*[25] egy UNIX rendszerhívás, mely segítségével memóriát tudunk foglalni, file-okat vagy eszközöket tudunk a memóriába mappelni.

A függvény szignatúrája:

```
void *
mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset);
```

Első paramétere – *void *addr* – egy cím, mely, ha nem *NULL*, akkor az egy javaslat a rendszer számára, hogy hova kérjük a mappelést, ha *NULL*, akkor a rendszer dönti el, hogy milyen címre rakja (ASLR-rel ezt kívánjuk első sorban randomizálni).

Második paramétere – *size_t len* – megadja, hogy mekkora a mappelendő címtér.

Harmadik paramétere – *int prot* – megadja, hogy milyen jogokat adunk az adott memória régiónak, ez lehet *PROT_NONE, PROT_READ, PROT_WRITE, PROT_EXEC* és ezek kombinációja. *mprotect(...)* esetén ezekre teszünk megszorításokat.

Negyedik paramétere – *int flags* – megadja a mappelt objektum típusát, opcióit és tulajdonságait.

Ötödik – *int fd* – egy file descriptor, mely megadja a mappelendő objektum descriptorát.

Hatodik paraméter – *off_t offset* – megadja, hogy az *fd*-ben milyen *offset*-től kezdve mappeljük be a memóriába a *len* paraméter által megadott hosszú régiót.

Az *mmap(...)* rendszerhívást kernel oldalon a *sys_mmap(...)* függvény valósítja meg, amit az ASLR implementálásakor módosítani fogok.

2.4.3 Trap

A trap egy speciális CPU szolgáltatás, mely segítségével bizonyos események következtében a rendszer kernel módba vált. A trap kiváltásának hatására a rendszer állapota elmentésre kerül. Egy részét maga a CPU menti le, más részét a trap-et lekezelő rutin.

A trap a folyamatok számára szinkron. Trap keletkezik zéró osztásnál, **laphibánál** és egyéb események bekövetkezésekor. Ez bővebben megtalálható az Intel SDM-ben[1].

Jelen mű esetében a laphiba esetén dobott trap lesz a fontos, mint ahogy ez később látható lesz.

2.4.4 Alacsony szintű memóriakezelő rutinok

A korábban leírt *MD* réteghez hozzátartozik amd64 architektúra esetén a *sys/amd64/amd64/support.S* file-ban található *copyin(...)* és *copyout(...)* függvénycsaládhoz tartozó függvények, melyeket a kernel space – user space határon történő adatmozgatásra használunk. A FreeBSD rendszerben minden ilyen kernel space - user space interakció ezen függvényekre van visszavezetve. Ami nem ezek segítségével történik, az programozási hiba és potenciális biztonsági probléma.

A függvénycsalád függvényei:

```
/* kernelből ki és bemásoló függvények */
int     copyinstr(const void * __restrict udaddr, void * __restrict kaddr,
                    size_t   len,   size_t   *   __restrict   lencopied)
__nonnull(1) __nonnull(2);

int     copyin(const void * __restrict udaddr, void * __restrict kaddr,
               size_t len) __nonnull(1) __nonnull(2);

int     copyin_nofault(const void * __restrict udaddr,
               void   *   __restrict   kaddr,   size_t   len)   __nonnull(1)
__nonnull(2);

int     copyout(const void * __restrict kaddr, void * __restrict udaddr,
               size_t len) __nonnull(1) __nonnull(2);
```

```
int     copyout_nofault(const void * __restrict kaddr,
                void   *   __restrict   udaddr,   size_t   len)   __nonnull(1)
__nonnull(2);

/* Fetch User Type */
int     fubyte(const void *base);
long    fuword(const void *base);
int     fuword16(void *base);
int32_t fuword32(const void *base);
int64_t fuword64(const void *base);

/* Set User Type */
int     subyte(void *base, int byte);
int     suword(void *base, long word);
int     suword16(void *base, int word);
int     suword32(void *base, int32_t word);
int     suword64(void *base, int64_t word);

/* Compare and Set User Type */
uint32_t  casuword32(volatile  uint32_t  *base,  uint32_t  oldval,  uint32_t
newval);

u_long   casuword(volatile u_long *p, u_long oldval, u_long newval);
```

Később ezek a függvények még elő fognak kerülni, ezért soroltam itt fel őket.

2.4.5 Sysctl interface

A sysctl[7][8][34] a FreeBSD kernel egy olyan szolgáltatása, mely segítségével a futó rendszer paramétereit privilegizált felhasználó képes megváltoztatni.

A *sysctl* parancs segítségével állítható értékek egy kernelben található memóriaterületre mutatnak.

2.4.6 Tunable interface

A tunable hasonló interface, mint a sysctl, azonban ez a szolgáltatás nem futás, , hanem boot időben érhető el. Értékeit a *loader*-ből kell megadni, vagy a *kenv* parancs segítségével az adott modul betöltése előtt.

A sysctl és a tunable interface mutathat egy közös memóriaterületre, ebben az esetben az adott kernel paraméter mind boot időben, mind futásidőben állítható.

2.4.7 CPU inicializáció

Minden operációs rendszer első feladata a processzor inicializálása, miután a BIOS-ról vagy firmware-ről a vezérlés rá került.

FreeBSD esetében sincs ez másképp. A CPU inicializáció FreeBSD alatt a *hammer_time(...)* függvény segítségével történik. A teljesség igénye nélkül pár funkció, amit ez a függvény végez: beállítja a kernel stacket, beállítja a descriptorokat, létrehozza az 0-ás process-t, amely a kernel-t írja le, inicializálja a kernel-t érintő mutex-eket, *identifikálja a processzort és inicializálja*, ezt követően engedélyezi processzor cache vonalakat, inicializálja a memóriát és elindítja a 0-ás process-t.

Számunkra a *hammer_time(...)* függvény által hívott *initializecpu(…)* függvény az érdekes, mivel ez végzi a valódi CPU inicializációt. Ez a függvény további részfüggvényekre bomlik, mely függvények CPUID alapján funkciókat kapcsolnak be a processzorban, például XD bit támogatást, az SMEP támogatást, vagy az általam készített patch után az SMAP támogatást.

2.4.8 Kernel debug

Mint minden software, a kernelek is tartalmaznak hibákat, azonban egy hiba keresése a kernelben részben eltér a megszokottól, mivel nincsenek meg a jól megszokott debuggolást segítő rendszerhívásaink, mint például a *ptrace* vagy a *procfs*[21].

2.4.8.1 DDB

A DDB[20] a FreeBSD kernel úgynevezett in-kernel interaktív debuggere. Ez annyit tesz, hogy a kernel binárisa tartalmaz egy alrendszert, mely bizonyos események hatására aktiválódik. Ilyen kiváltó esemény lehet többek között egy kernel panic vagy egy sysctl állítása.

A DDB azért hasznos, mert a kernel hibakeresése nem igényel semmilyen külső programot, így *panic(…)* esetén is – amikor minden egyéb folyamatot felfüggeszt a kernel – utána tudunk nézni a hiba okának.

A DDB egy igen nagy tudású eszköz, mely tulajdonságait a GDB után modellezték. Legnagyobb előnye, hogy scriptelhető, így ha a fejlesztőknek szükségük van bizonyos információkra, melyek csak bonyolult lépések sorozataként érhetőek el, akkor ezt csak egyszer kell megírnunk és utána a scriptet futtatnunk a debuggeren belül.

2.4.8.2 KGDB

A FreeBSD rendszerekben a másik szállított debugger a *kgdb*[22], mely a *gdb* egy kiterjesztése kernel debuggoláshoz. Ezt az eszközt leginkább post-mortem esetekben használjuk, de on-line kernel is debuggolható vele (on-line esetben *kgdb /boot/kernel/kernel /dev/mem* parancs futtatása szükséges).

A *gdb* teljes funkcionalitását tartalmazza kiegészítve pár kernel specifikus kiegészítéssel.

2.5 Intel x86-64

2.5.1 Architektúra

Az x86-64 az x86 (eredeti nevén: Intel 8086) leszármazottja, melyet az AMD fejlesztett ki. Legfontosabb újítás benne, hogy a processzor 64 bites regisztereket kapott, 64 bites virtuális memóriakezeléssel lett bővítve, a szegmentáció meg lett szüntetve. A regiszterek számát megnövelték és bevezették az NX (XD) technológiát.

A processzor egy kompatibilitási módot is tartalmaz, amikor 32 bites módban fut, és a régi technológiák ekkor működnek, miközben az újaknak csak részhalmazai.

Az x86-64-es CPU-ra a gyártók eltérően referálhatnak: Microsoft esetén az *x64* a használatos, BSD rendszerek esetében az *amd64*, az Intel pedig a dokumentációban rendszerint *IA-32e, EM64T* vagy *Intel 64* megnevezéssel illeti a technológiát. Én a továbbiakban az *amd64* és *x86-64* megnevezéseket fogom használni, mivel a FreeBSD esetében ezek az elterjedtek.

2.5.2 Bináris patchelés – önmódosító kód

Bináris patchelés alatt értendő az a folyamat, amikor a futó kód saját magát módosítja.

Általában teljesítmény okokból alkalmazzák. Esetemben a patchelendő utasítások számára fent van tartva multi byte *NOP* szekvenciákkal a hely, és ha a feltételnek megfelel, akkor ezek a szekvenciák ki lesznek cserélve a cél szekvenciákra.

2.5.3 XD bit

Az újabb Intel és AMD processzorokban alkalmazott technológia, mellyel lapokat nem futtathatónak lehet jelölni.

Az *XD* feloldása eXecution Disable. AMD esetében az *XD*-t *NX*-nek nevezik, mely feloldása Never eXecute.

2.5.4 SMAP

Az SMAP[2][3] az Intel egyik legújabb fejlesztése, mely egy biztonsági kiegészítés. Feloldása a Supervisor Mode Access Prevention.

A megoldás lényegében az UDEREF[15] hardware oldali[19] megoldása.

A technológia megakadályozza, hogy lapozás esetén supervisor privilégiumban futó kód user módú kódhoz tartozó memóriaterülethez férjen hozzá. Ha ezt megszegi, akkor laphiba kivétel dobódik. Ezt a korlátozást átmenetileg ki lehet kapcsolni az *EFLAGS.AC* flag ki vagy be billentésével. Erről bővebben a dolgozat hátralevő részében lesz szó.

2.5.5 CPL

Current privilege level, aktuális privilégium szint. Az Intel processzorokban különböző privilégiumszintek vannak, ezeket hívjuk Ring-nek is. Négy szint definiált. A 0, 1, 2 supervisor módú hozzáférést biztosít, 0 biztosítja a legtöbb jogot és növekedve csökken a jogosultságok köre. A Ring 3 pedig a user módú hozzáférés.

Az operációs rendszerek tipikusan csak a 0-ás és a 3-as módot implementálják.

3 Megvalósítás

3.1 SMAP implementálása FreeBSD kernelben

3.1.1 ksp_kpatch – futásidejű bináris patchelés

3.1.1.1 Ismertetés

Mivel a FreeBSD alapból nem támogatja framework-szerűen a bináris patchelés folyamatát, így nekem kellett implementálnom a szakdolgozat keretein belül. Az így elkészített *ksp_kpatch* framework-öt részben a Linux kernelben található *alternatives[18]* framework inspirálta, részben a józan ész.

A framework szükségességét az indokolja, hogy szép és hatékony megoldást kerestem a forrásban elszórtan és változó számban előforduló, futásidőben bizonyos feltételek alapján kicserélendő assembly utasítások cseréjére. Ezt a framework nélkül is meg lehetett volna oldani feltételek beiktatásával, de mint azt már többször is hangsúlyoztam, FreeBSD esetében az elsődleges szempontok között a teljesítmény áll, így egy rendszer mélyét érintő és sokat használt függvénycsaládot befolyásoló kiegészítésnek hatékonynak kell lennie. A nem hatékony megoldások a következők lennének: minden egyes feltételhez kötött utasítás lefutása előtt meg kellene vizsgálni az adott feltételt és a feltételnek megfelelően az adott utasításra ugrani. Ennek költsége, mint látható elég nagy, mivel tartalmaz egy feltételt és egy ugrást. A másik megoldás esetében részben önmódosító kód van alkalmazva, amikor egy alternatív utasítást megelőz és követ egy-egy ugrás. Mint látható, ez sem egy hatékony megoldás, több szempontból sem; ha a cím, amire ugrani kell, nem egy lapon van azzal a címmel, ahol a megelőző utasítás szerepelt, akkor laphiba keletkezhet és a lapot be kell hozni, ergo a költség hatalmas lesz. Ezen tényezők miatt a futásidejű bináris patchelés mellett döntöttem. Ez azt takarja, hogy amikor a rendszer bootolás közben a CPU-t inicializálja, akkor bizonyos feltételek és egy speciális adatszerkezet segítségével megkeresi a kernel binárisában található utasításokat és kicseréli azokat másikra. Ennek a megoldásnak is vannak hátrányai, de jóval kevesebb, mint az előzőeknek. Ez a megvalósítás *NOP* utasításokat használ az alternatív utasítások helyének biztosítására, melyeket a patchelés során kicserélek a cél utasításokra. A *NOP* utasítás nem csak 1 byte-os *NOP* lehet,

hanem úgynevezett *multi byte NOP* is, melyeket az Intel SDM tartalmaz. A *multi byte NOP* utasítások lényege, hogy maga az utasítás több byte hosszú, de atomian egy órajel alatt futnak le.

A megvalósítás átnézésében és finomításában segítséget kaptam Konstantin Belousov FreeBSD fejlesztőtől is. A szükséges ismeretek nagy részét az Intel SDM tartalmazza, részben pedig a GNU assembler dokumentációja.

A megvalósítás jelenleg csak a szükséges funkcionalitást tartalmazza, azaz jelen formájában csak a kernel image patchelése lehetséges vele, a kernel moduloké nem. Ez a fejlesztői döntés azért lett meghozva, mert nem kapcsolódik közvetlenül a biztonsági témakörhöz. Azonban szükség volt rá, hogy az SMAP technológiára felkészített kernel képes legyen felbootolni olyan rendszeren újrafordítás nélkül is, amely CPU-ja nem tartalmaz SMAP technológiát.

Konstantinnal ebből egy kisebb vita alakult ki, hogy a hely fenntartásra szolgáló *NOP* utasítások lassítják a rendszert, így az SMAP patchből két verzió készült el; egy, amely esetében csak fordítás időben lehet az alternatív utasításokat a kernelbe fordítani és egy, amely a *ksp_kpatch* frameworköt használja.

3.1.1.2 Adatstruktúra

A használt speciális struktúra a következő - *sys/amd64/include/ksp_kpatch.h*:

```
struct ksp_kpatch {
        char    *patchable_address;     /* 64 bit */
        char    *patch_address;         /* 64 bit */
        u_int   feature_bit;            /* 32 bit */
        u_short feature_selector;       /* 16 bit */
        u_char  patchable_size;         /*  8 bit */
        u_char  patch_size;             /*  8 bit */
};
```

Az első struktúra elem – *char *patchable_address* – megadja a kernel image-ben a lecserélendő utasítások címét. Az ezt követő elem – *char *patch_address* – megadja a lecserélő utasítások címét. Az *u_int feature_bit* megadja, hogy az *u_short feature_selector* által kiválasztott státusz byte melyik bitje alapján kell lecserélni vagy nem lecserélni az adott utasítás sorozatot. Az utolsó két elem pedig megadja a patchelendő és a patch utasítássorozatok méretét. Erre a későbbiekben lesz szükség, mivel ellenőrzöm, hogy a patch nem nagyobb-e, mint a patchelendő. Ha igen, akkor hibát dobok – konkrétan egy KASSERT van alkalmazva, így hiba esetén kernel *panic(...)* lesz a végeredmény.

3.1.1.3 ksp_kpatch létrehozása

A kernel forrásban *ksp_kpatch*-et a következő formában lehet létrehozni:

```
#define _ksp_kpatch_clac                                                \
89071:                                                                  \
        .byte 0x0f,0x1f,0x00 ;                  /* patchable - nop */   \
89072:                                                                  \
        .pushsection set_ksp_kpatch_set, "a" ;                          \
                .quad   89071b ;                /* &patchable */        \
                .quad   89073f ;                /* &patch */            \
                .int    CPUID_STDEXT_SMAP ;     /* feature_bit*/        \
                .word   CPU_STDEXT_FEATURE ;                            \
                .byte   89072b-89071b ;                                 \
                .byte   89074f-89073f ;                                 \
        .popsection ;                                                   \
        .pushsection set_ksp_kpatch_patch_set, "ax" ;                   \
89073:                                                                  \
                .byte 0x0f,0x01,0xca ;          /* patch - clac */      \
89074:                                                                  \
        .popsection
```

Mint ahogy az látható, a munka javarészét a fordítóval végeztetem el. Az előbb bemutatott struktúra mezői itt lesznek kitöltve. Ehhez a GNU assembler *.pushsection* és *.popsection* között található részek másik szekcióba kerülnek a binárison belül, így a patch használati helyen a binárisban csak a *89071* és *89072* címkék közötti bináris szekvencia fog szerepelni, a többi "meta adat" a kernel image más részében fog elhelyezkedni.

A *.set_ksp_kpatch_set* szekcióban találhatók a *ksp_kpatch* struktúra adatai, az egyetlen figyelemre méltó dolog itt a méret számítások, melyeket a címkék egymásból kivonásával kapjuk meg.

A *.set_ksp_kpatch_patch_set* szekcióban találhatók a patch byte sorozatok. A szekciókat linkert bemutató részben bővebben kifejtem.

3.1.1.4 linker script módosítások

A módosítandó file a *sys/conf/ldscript.amd64*, mely a kernel bináris linkelését írja le. A módosítást most patch formátumban szemléltetem (mi változott az előző verzióhoz képest).

```
diff --git a/sys/conf/ldscript.amd64 b/sys/conf/ldscript.amd64
index 9210a73..3f22e63 100644
--- a/sys/conf/ldscript.amd64
+++ b/sys/conf/ldscript.amd64
@@ -67,6 +67,18 @@ SECTIONS
   PROVIDE (__etext = .);
   PROVIDE (_etext = .);
   PROVIDE (etext = .);
```

```
+  set_ksp_kpatch_set   :
+  {
+    PROVIDE ( __start_set_ksp_kpatch_set = . );
+    KEEP (*(set_ksp_kpatch_set));
+    PROVIDE ( __stop_set_ksp_kpatch_set = . );
+  }
+  set_ksp_kpatch_patch_set     :
+  {
+    PROVIDE ( __start_set_ksp_kpatch_patch_set = . );
+    KEEP (*(set_ksp_kpatch_patch_set));
+    PROVIDE ( __stop_set_ksp_kpatch_patch_set = . );
+  }
   .rodata         : { *(.rodata .rodata.* .gnu.linkonce.r.*) }
   .rodata1        : { *(.rodata1) }
   .eh_frame_hdr : { *(.eh_frame_hdr) }
```

Itt látható, hogy a hozzáadott szekciók még két speciális elemet tartalmaznak, a *__start_set_ksp_kpatch_patch_set* és *__stop_set_ksp_kpatch_patch_set* változókat, melyek segítségével C kódból egyszerűen meg tudom találni az adott szekció által tartalmazott adatokat – jelen esetben ez egy tömb a *set_ksp_kpatch_set* esetében és egy változó hosszúságú tömb a *set_ksp_kpatch_patch_set* szekció esetén.

3.1.1.5 Boot idejű patchelés folyamata

A megvalósítás a *sys/amd64/amd64/ksp_kpatch.c* file-ban található. Itt három függvény található:

```
static void
ksp_pad_with_nops(void *_buffer, unsigned int len)
{
     ...
}
```

Ez a funkció abban az esetben van használva, ha a patch mérete kisebb, mint a patchelendő kód mérete. Ebben az esetben a *ksp_pad_with_nops(...)* funkció kiválasztja a megfelelő padding *NOP* utasításokat. Ez alatt azt kell érteni, hogy a lehető legnagyobb atomi *NOP* utasításokkal tölti fel a patchelendő üres helyet, hogy a futást a lehető legkisebb mértékben lassítsa.

```
void
ksp_kpatch_apply(struct ksp_kpatch *patch)
{
     ...
}
```

A *ksp_kpatch_apply(...)* végzi a valódi patchelést. A fent definiált struktúrát ez a funkció használja. Kiválasztja a megfelelő státusz byte-okat és ellenőrzi a *feature_bit* alapján, hogy szükséges-e az adott kódrészlet kicserélésére. Ha szükség van rá, akkor a

megfelelő helyen felülírja az utasításokat. Természetesen előtte ellenőrzi, hogy a méretek megfelelőek-e.

```
void
ksp_kpatch(void)
{
     ...
}
```

A *ksp_kpatch(…)* keresi ki a bináris image-ből a *ksp_kpatch*-eket tartalmazó részt, és az ott található meta adatok alapján végigiterál rajtuk. A megtaláláshoz itt használjuk fel a linker szekció leírásban korábban említett két speciális változót (*__start_set_ksp_kpatch_patch_set* és *__stop_set_ksp_kpatch_patch_set*).

3.1.1.6 boot folyamatba beillesztés

A *ksp_kpatch(…)* a *sys/amd64/amd64/machdep.c* file-ban található *hammer_time(…)* funkcióban kap helyet. A *hammer_time(…)* függvény felelős felkészíteni a CPU-t UNIX rendszer futtatására.

```
@@ -1800,6 +1801,9 @@ hammer_time(u_int64_t modulep, u_int64_t physfree)
        initializecpu();        /* Initialize CPU registers */
        initializecpucache();

+       /* patching kernel text for new instructions */
+       ksp_kpatch();
+
        /* doublefault stack space, runs on ist1 */
        common_tss[0].tss_ist1                                          =
(long)&dblfault_stack[sizeof(dblfault_stack)];
```

A patchelés a CPU és CPU cache inicializáció után történik meg. A cache törlésével nem kell foglalkozni[4][5].

3.1.2 SMAP

Az Intel 64 (X86-64) architektúra eddig két megoldást adott a lineáris memóriából utasítás végrehajtás korlátozására: az eXecute Disable (XD, NX) bitet és az SMEP technológiát.

A harmadik ilyen szabályozó technológia az SMAP[2], mely az adathozzáférést hivatott szabályozni. Az SMAP feloldása Supervisor Mode Access Prevention.

Ha az SMAP be van kapcsolva, akkor a processzor elutasítja a supervisor módú adathozzáférést azon lapokhoz, melyek user módón elérhetők. Az elutasított hozzáférés laphibát eredményez. Ez azonban nem minden esetben kívánatos, mivel a kernelnek

hozzá kell férnie user lapokhoz – például a korábban említett copyin és copyout függvénycsalád tagjainak – mivel ez a természetes funkcionalitáshoz hozzá tartozik. Erre az esetre az SMAP architektúra megengedi, hogy átmenetileg hozzáférjünk user space lapokhoz: engedélyezésre a *stac*, visszavonásra a *clac* utasítás szolgál, melyekkel az *EFLAGS.AC* bitet tudjuk állítani.

3.1.2.1 A cpuid

Az SMAP támogatás megléte a *cpuid* utasítással kérhető le. Ha a processzor támogatja, akkor *CPUID.(EAX=07H,ECX=0H):EBX.SMAP[bit 20] = 1*.

Ezt követően a *CR4.SMAP[bit 21]* bit beállításával tudjuk bekapcsolni a szolgáltatást. A szolgáltatás kikapcsolt paging módban is bekapcsolható, azonban nincs hatással a processzor ezen módú működésére.

3.1.2.2 STAC

SMAP védelem alatt átmeneti user space hozzáférés engedélyezése.

EFLAGS.AC bit beállítása, csak Ring 0-ban lehet használni.

gépi kódja: *0F 01 CB*

3.1.2.3 CLAC

SMAP védelem alatt átmeneti user space hozzáférés visszavonása.

EFLAGS.AC bit törlése, csak Ring 0-ban lehet használni.

gépi kódja: *0F 01 CA*

3.1.2.4 SMAP és a hozzáférési jogok

- Adat olvasás engedélyezett[2]:
  - Ha *CR4.SMAP = 0*, akkor adat olvasható minden lineáris címről, amihez tartozik valid címfordítás
  - Ha a *CR4.SMAP = 1*, akkor a hozzáférési jogok függenek a *CPL*-től és az *EFLAGS.AC* bittől:
    * ha *CPL < 3* és *EFLAGS.AC = 1*: adat olvasható minden lineáris címről, amihez tartozik valid címfordítás

* ha *CPL = 3* vagy *EFLAGS.AC = 0*, akkor adat olvasható minden olyan lineáris címről, melyhez tartozik valid címfordítás mely esetében a *U/S flag (bit 2) = 0* legalább egy címfordítást felügyelő lapbejegyzésen

- Adat írás engedélyezett:
  - Ha a *CR0.WP = 0* és *CR4.SMAP = 0*, akkor adat írható minden lineáris címre, amihez tartozik valid címfordítás
  - Ha a *CR0.WP = 0* és *CR4.SMAP = 1*, akkor a hozzáférés függ a *CPL*-től és *EFLAGS.AC* flag-től.
    * Ha *CPL < 3* és *EFLAGS.AC = 1*, akkor adat írható minden lineáris címre, amihez tartozik valid címfordítás
    * Ha *CPL = 0* vagy *EFLAGS.AC = 0*, akkor adat írható minden olyan lineáris címen, amelyhez tartozik valid címfordítás mely esetében a *U/S flag (bit 2) = 0* legalább egy lapbejegyzésen
  - Ha a *CR0.WP = 1* és *CR4.SMAP = 0*, akkor adat írható minden olyan lineáris címre, amihez tartozik valid címfordítás mely esetében a *R/W flag (bit 1) = 1* minden címfordítást felügyelő lapbejegyzésen
- Ha a *CR0.WP = 1* és *CR4.SMAP = 1*, akkor a hozzáférési jogok függenek a *CPL*-től és az *EFLAGS.AC* flag-től:
  - Ha a *CPL < 3* és *EFLAGS.AC = 1*, akkor adat írható minden olyan lineáris címre, amihez tartozik valid címfordítás mely esetében a *R/W flag (bit 1) = 1* minden címfordítást felügyelő lapbejegyzésen
  - Ha a *CPL = 3* vagy *EFLAGS.AC = 0*, akkor adat írható minden olyan lineáris címre, melyhez tartozik valid címfordítás mely esetében a *U/S flag (bit 2) = 0* legalább egy címfordítást felügyelő lapbejegyzésen és a címfordítást felügyelő összes lapbejegyzésen az *R/W (bit 1) = 1*

Az SMAP által nem engedélyezett adathozzáférések laphibát eredményeznek. Az SMAP nincs hatással az instruction fetch-re, a user módú adat elérésre és a supervisor módú adatelérésre csak supervisor lapokra.

3.1.2.5 SMAP által kiváltott laphiba

Az SMAP által kiváltott kivétel laphiba és a hibakódja 1, ha olvasásról van szó, és 3, ha írásról van szó.

3.1.3 Szükséges kernel módosítások

A következőkben a Konstantin által javasolt fordítás idejű változatot mutatom be. A boot folyamat alatt önmódosító változat annyiban tér el, hogy a *STAC* és *CLAC* makró alatt nem maga a stac és clac byte sorozat található, hanem a *ksp_kpatch* rész alatt bemutatott *ksp_kpatch*, mely tartalmazza a két utasítást.

3.1.3.1 initcpu.c

Az SMAP technológiát az SMEP-hez hasonlóan nem az initcpu szakaszban kapcsoljuk be, mivel a bootoláskor problémák léphetnek fel - mivel a *loader* user flag-et állít be a lapjain - hanem a pmap alrendszerben késleltetve, a kernel táblákra áttérés után.

```
 diff --git a/sys/amd64/amd64/initcpu.c b/sys/amd64/amd64/initcpu.c
index 4abed4c..1f0ae6c 100644
--- a/sys/amd64/amd64/initcpu.c
+++ b/sys/amd64/amd64/initcpu.c
@@ -165,13 +165,17 @@ initializecpu(void)
                cr4 |= CR4_FSGSBASE;

        /*
-        * Postpone enabling the SMEP on the boot CPU until the page
-        * tables are switched from the boot loader identity mapping
-        * to the kernel tables.  The boot loader enables the U bit in
-        * its tables.
+        * Postpone enabling the SMEP and the SMAP on the boot CPU until
+        * the page tables are switched from the boot loader identity
+        * mapping to the kernel tables.
+        * The boot loader enables the U bit in its tables.
         */
        if (!IS_BSP() && (cpu_stdext_feature & CPUID_STDEXT_SMEP))
                cr4 |= CR4_SMEP;
+#ifdef INTEL_SMAP
+       if (!IS_BSP() && (cpu_stdext_feature & CPUID_STDEXT_SMAP))
+               cr4 |= CR4_SMAP;
+#endif /* INTEL_SMAP */
        load_cr4(cr4);
        if ((amd_feature & AMDID_NX) != 0) {
                msr = rdmsr(MSR_EFER) | EFER_NXE;
```

3.1.3.2 pmap.c

Az SMAP support-ot az *initcpu* helyett a *pmap* alrendszerben kapcsoljuk be, mivel a bootolás korai szakaszában még nincs rá szükség és problémákat is okozhat.

```
diff --git a/sys/amd64/amd64/pmap.c b/sys/amd64/amd64/pmap.c
index 1b1c86c..8fb223a 100644
--- a/sys/amd64/amd64/pmap.c
+++ b/sys/amd64/amd64/pmap.c
@@ -98,6 +98,7 @@ __FBSDID("$FreeBSD$");
  *     and to when physical maps must be made correct.
  */

+#include "opt_cpu.h"
 #include "opt_pmap.h"
 #include "opt_vm.h"

@@ -665,6 +666,18 @@ pmap_bootstrap(vm_paddr_t *firstaddr)
        if (cpu_stdext_feature & CPUID_STDEXT_SMEP)
                load_cr4(rcr4() | CR4_SMEP);

+       if (cpu_stdext_feature & CPUID_STDEXT_SMAP)
+#ifdef INTEL_SMAP
+               load_cr4(rcr4() | CR4_SMAP);
+       else
+               panic("The kernel compiled with \"options INTEL_SMAP\","
+                               "but your CPU doesn't support SMAP!\n");
+#else  /* !INTEL_SMAP */
+               printf("Your CPU has support for SMAP security feature. "
+                       "You should recompile the kernel with "
+                       "\"options INTEL_SMAP\" to use them.\n");
+#endif /* INTEL_SMAP */
+
        /*
         * Initialize the kernel pmap (which is statically allocated).
         */
```

3.1.3.3 trap.c

Az SMAP leírásnál ismertetett *#PF* laphibát kell lekezelni a *sys/amd64/amd64/trap.c* file-ban található *trap(...)* függvényben. Ha az SMAP nincs bekapcsolva, akkor a laphibával SMAP részről nem foglalkozunk. Ha be van kapcsolva, akkor folytatjuk az ellenőrzést: ha user módban vagyunk vagy az *EFLAGS.AC* értéke 1, akkor ugyancsak nem foglalkozunk a laphibával SMAP részről, mert ekkor vagy nem vonatkozik ránk vagy engedélyezett. Minden más esetben le kell kezelnünk a laphibát SMAP szempontjából, azaz kernel *panic(...)* lesz a végeredmény.

Az előbbi feltételek ellenőrzésére egy külön static függvényt vezettem be, mivel így olvashatóbb a kód.

```
diff --git a/sys/amd64/amd64/trap.c b/sys/amd64/amd64/trap.c
index 6fcca81..0afc891 100644
--- a/sys/amd64/amd64/trap.c
+++ b/sys/amd64/amd64/trap.c
@@ -127,6 +127,9 @@ void dblfault_handler(struct trapframe *frame);

 static int trap_pfault(struct trapframe *, int);
 static void trap_fatal(struct trapframe *, vm_offset_t);
```

```
+#ifdef INTEL_SMAP
+static bool smap_access_violation(struct trapframe *, int usermode);
+#endif

 #define MAX_TRAP_MSG           33
 static char *trap_msg[] = {
diff --git a/sys/amd64/amd64/trap.c b/sys/amd64/amd64/trap.c
index 6fcca81..0afc891 100644
--- a/sys/amd64/amd64/trap.c
+++ b/sys/amd64/amd64/trap.c
@@ -127,6 +127,9 @@ void dblfault_handler(struct trapframe *frame);

 static int trap_pfault(struct trapframe *, int);
 static void trap_fatal(struct trapframe *, vm_offset_t);
+#ifdef INTEL_SMAP
+static bool smap_access_violation(struct trapframe *, int usermode);
+#endif

 #define MAX_TRAP_MSG           33
 static char *trap_msg[] = {
@@ -718,6 +721,14 @@ trap_pfault(frame, usermode)

                map = &vm->vm_map;

+#ifdef INTEL_SMAP
+              if (__predict_false(smap_access_violation(frame, usermode)))
{
+                       trap_fatal(frame, eva);
+                       return (-1);
+               }
+#endif
+
                /*
                 * When accessing a usermode address, kernel must be
                 * ready to accept the page fault, and provide a
@@ -874,6 +884,20 @@ trap_fatal(frame, eva)
                panic("unknown/reserved trap");
 }

+#ifdef INTEL_SMAP
+static bool
+smap_access_violation(struct trapframe *frame, int usermode)
+{
+       if ((cpu_stdext_feature & CPUID_STDEXT_SMAP) == 0)
+               return (false);
+
+       if (usermode || (frame->tf_rflags & PSL_AC) != 0)
+               return (false);
+
+       return (true);
+}
+#endif
+
 /*
  * Double fault handler. Called when a fault occurs while writing
  * a frame for a trap/exception onto the stack. This usually occurs
```

3.1.3.4 support.S

A *sys/amd64/amd64/support.S* file-ban történő módosítások kivétel nélkül a user space memóriát legálisan elérő utasítások *stac* és *clac* utasításokkal való körülvétele. Ezzel átmenetileg kikapcsoljuk az SMAP-et, így a kernel képes hozzáférni user space memóriához.

```
diff --git a/sys/amd64/amd64/support.S b/sys/amd64/amd64/support.S
index 77dbf63..9c12087 100644
--- a/sys/amd64/amd64/support.S
+++ b/sys/amd64/amd64/support.S
@@ -35,6 +35,7 @@
 #include <machine/asmacros.h>
 #include <machine/asmacros.h>
 #include <machine/intr_machdep.h>
 #include <machine/pmap.h>
+#include <machine/smap_instr.h>

 #include "assym.s"

@@ -244,12 +245,16 @@ ENTRY(copyout)

        shrq    $3,%rcx
        cld
+       STAC
        rep
        movsq
+       CLAC
        movb    %dl,%cl
        andb    $7,%cl
+       STAC
        rep
        movsb
+       CLAC

 done_copyout:
        xorl    %eax,%eax
@@ -290,12 +295,16 @@ ENTRY(copyin)
        movb    %cl,%al
        shrq    $3,%rcx                         /* copy longword-wise */
        cld
+       STAC
        rep
        movsq
+       CLAC
        movb    %al,%cl
        andb    $7,%cl                          /* copy remaining bytes */
+       STAC
        rep
        movsb
+       CLAC
```

Ez egy példa a sok függvény közül, melyet a *support.S* tartalmaz, a többi esetben is hasonló módosítások vannak. Ezeken felül, ha valamilyen exception történne,

akkor abban az esetben a *EFLAGS.AC* flag-et törölni kell, hogy a control-flow-ban ne terjedjen random helyekre és ne okozzon nem kívánt sérülékenységet.

3.1.3.5 exception.S

Az előző alpont alatt említett kivételkezelés esetén kezelendő megvalósítás. (Az *EFLAGS* értéke kivétel hatására automatikusan a stack-re kerül.)

```
diff --git a/sys/amd64/amd64/exception.S b/sys/amd64/amd64/exception.S
index 89ad638..2956efc 100644
--- a/sys/amd64/amd64/exception.S
+++ b/sys/amd64/amd64/exception.S
@@ -42,6 +42,7 @@
 #include <machine/asmacros.h>
 #include <machine/psl.h>
 #include <machine/trap.h>
+#include <machine/smap_instr.h>
 #include <machine/specialreg.h>

 #include "assym.s"
@@ -196,6 +197,7 @@ alltraps_pushregs_no_rdi:
         movq    %r15,TF_R15(%rsp)
         movl    $TF_HASSEGS,TF_FLAGS(%rsp)
         cld
+       CLAC
         FAKE_MCOUNT(TF_RIP(%rsp))
 #ifdef KDTRACE_HOOKS
         /*
@@ -276,6 +278,7 @@ IDTVEC(dblfault)
         movw    %ds,TF_DS(%rsp)
         movl    $TF_HASSEGS,TF_FLAGS(%rsp)
         cld
+       CLAC
         testb   $SEL_RPL_MASK,TF_CS(%rsp) /* Did we come from kernel? */
         jz       1f                            /* already running with kernel
GS.base */
         swapgs
@@ -379,6 +382,7 @@ IDTVEC(fast_syscall)
         movq    %r15,TF_R15(%rsp)       /* C preserved */
         movl    $TF_HASSEGS,TF_FLAGS(%rsp)
         cld
+       CLAC
         FAKE_MCOUNT(TF_RIP(%rsp))
         movq    PCPU(CURTHREAD),%rdi
         movq    %rsp,TD_FRAME(%rdi)
@@ -474,6 +478,7 @@ IDTVEC(nmi)
         movw    %ds,TF_DS(%rsp)
         movl    $TF_HASSEGS,TF_FLAGS(%rsp)
         cld
+       CLAC
         xorl    %ebx,%ebx
         testb   $SEL_RPL_MASK,TF_CS(%rsp)
         jnz     nmi_fromuserspace
@@ -533,6 +538,7 @@ nmi_calltrap:

         shrq    $3,%rcx         /* trap frame size in long words */
         cld
```

```
+       CLAC
        rep
        movsq                   /* copy trapframe */
```

3.1.3.6 asmacros.h

Itt a *PUSH_FRAME* makrót egészítettem, ki a *CLAC* utasítással, hogy kivétel esetén töröljük az *EFLAGS.AC* flag-et biztonsági okokból a trap kezelésekor. A helyreállított task-ra nincs hatással, mivel az *EFLAGS*-et még a törlés előtt elmenti a CPU a stack-re.

```
diff --git a/sys/amd64/include/asmacros.h b/sys/amd64/include/asmacros.h
index 1fb592a..c985623 100644
--- a/sys/amd64/include/asmacros.h
+++ b/sys/amd64/include/asmacros.h
@@ -167,7 +167,8 @@
        movw    %es,TF_ES(%rsp) ;                                       \
        movw    %ds,TF_DS(%rsp) ;                                       \
        movl    $TF_HASSEGS,TF_FLAGS(%rsp) ;                            \
-       cld
+       cld ;                                                           \
+       CLAC

 #define POP_FRAME                                                      \
        movq    TF_RDI(%rsp),%rdi ;                                     \
```

3.1.3.7 Módosítások összegzése

```
sys/amd64/amd64/exception.S     |  6 ++++++
 sys/amd64/amd64/identcpu.c      | 28 ++++++++++++++++++++---
 sys/amd64/amd64/initcpu.c       | 12 +++++++----
 sys/amd64/amd64/pmap.c          | 13 ++++++++++
 sys/amd64/amd64/support.S       | 48 ++++++++++++++++++++++++++++++++++++++++
 sys/amd64/amd64/trap.c          | 24 ++++++++++++++++++++
 sys/amd64/ia32/ia32_exception.S |  1 +
 sys/amd64/include/asmacros.h    |  3 ++-
 sys/amd64/include/cpufunc.h     | 23 ++++++++++++++++++++
 sys/amd64/include/smap_instr.h  | 14 ++++++++++++
 sys/conf/NOTES                  |  4 ++++
 sys/conf/options.amd64          |  3 +++
 sys/x86/include/psl.h           |  2 +-
 sys/x86/include/specialreg.h    |  1 +
 14 files changed, 173 insertions(+), 9 deletions(-)
```

3.1.4 Tesztek

Mivel az SMAP képes CPU-k csak júniusban jelennek meg a piacon, így tesztelni csak emulátoron (*Qemu 1.4* és újabbak támogatják bekapcsolt *TCG* esetén) tudtam. Júniusban hozzáférést kapok az Intel támogatásával SMAP képes géphez, és akkor teljesítményteszteket is végzek.

Az SMAP technológia igen egyszerűen tesztelhető funkcionalitás szempontjából. A scenario annyiból áll, hogy egy valid user space címet kell dereferálnunk kernelből, anélkül, hogy hozzáférést engedélyeztnük volna a kernelnek.

Erre készítettem[23] egy tesztesetet, mely egy FreeBSD *char device*-ból és egy user space programból áll. A user space program a stack-jén létrehoz egy tömböt, melynek kezdőcímét beleírja a *char dev*-be. Ezután, amikor a *char dev*-re read rendszerhívás érkezik, akkor derefeálja a kernel a user space processz memóriaterületét. Ekkor bekapcsolt és működő SMAP esetén a kernel *panic(...)*-ot dob.

Figure 4: SMAP akcióban - működő védelem

Ha a teszteset olyan gépen van lefuttatva, amely nem SMAP képes, akkor a user space memória tartalma mindenféle korlátozás nélkül kiolvasható.

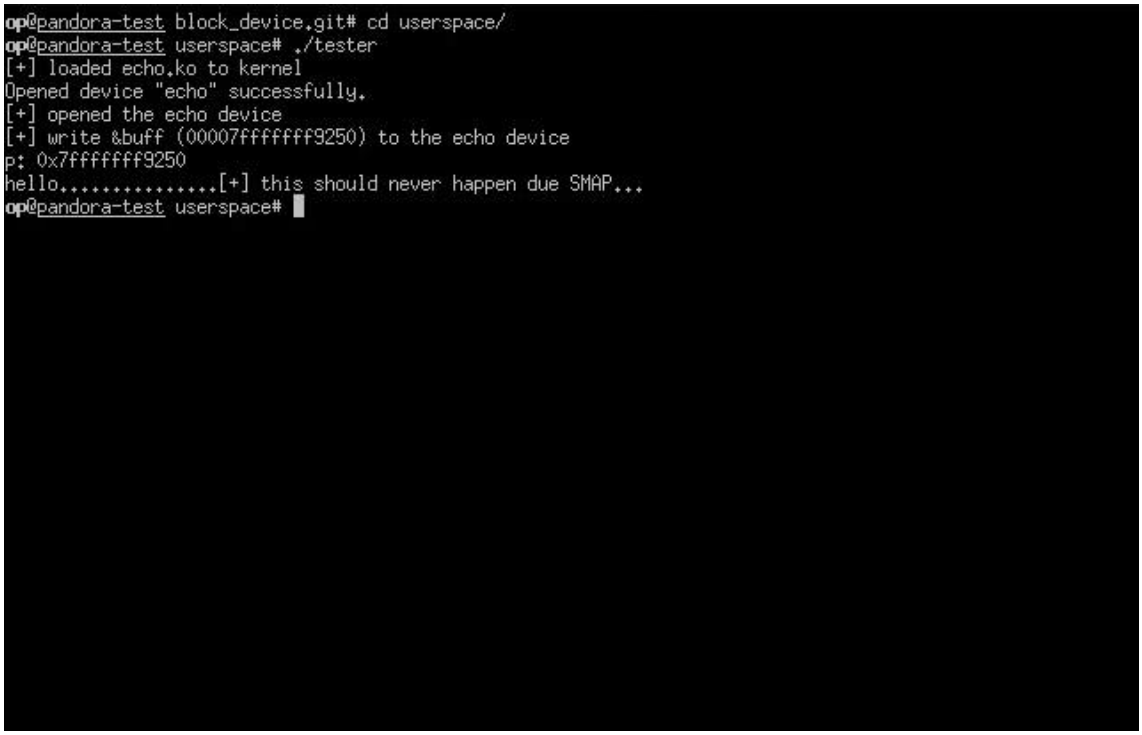


Figure 5: teszteset SMAP támogatás nélkül - a stack kiolvasható

3.2 ASLR megvalósítása FreeBSD-ben

3.2.1 ASLR

Az ASLR[10] megoldás célját és hátterét a 2.3.3 fejezetben már kifejtettem. Ebben a fejezetben a megvalósításra helyezem a hangsúlyt.

Az ASLR-ből jelen dolgozat keretén belül a stack randomizáció és az mmap randomizáció lett elkészítve.

3.2.2 Szükséges módosítások

3.2.2.1 vmspace

A *sys/vm/vm_map.h* file-ban található *vmspace* struktúra írja le az egy adott process-hez tartozó címteret. ASLR esetében egy adott process-hez mindig ugyanazt a randomot használom, ez leginkább sebesség okokra vezethető vissza.

Ehhez szükség volt két új elem bevezetésére: a *vm_aslr_delta_mmap* az *mmap(...)* rendszerhívásra alkalmazott memória címre alkalmazandó randomot tartalmazza, míg a *vm_aslr_delta_stack* a stack-re alkalmazandót.

```
diff --git a/sys/vm/vm_map.h b/sys/vm/vm_map.h
index 135b555..4fe05d6 100644
--- a/sys/vm/vm_map.h
+++ b/sys/vm/vm_map.h
@@ -240,6 +240,8 @@ struct vmspace {
        caddr_t vm_taddr;       /* (c) user virtual address of text */
        caddr_t vm_daddr;       /* (c) user virtual address of data */
        caddr_t vm_maxsaddr;    /* user VA at max stack growth */
+       vm_size_t vm_aslr_delta_mmap;   /* mmap() random delta for ASLR */
+       vm_size_t vm_aslr_delta_stack;  /* stack random delta for ASLR */
        volatile int vm_refcnt; /* number of references */
        /*
         * Keep the PMAP last, so that CPU-specific variations of that
```

3.2.2.2 vm_map

A *sys/vm/vm_map.c* file-ban található *vmspace_alloc(...)* függvényben történt változtatás a megfelelő inicializációhoz kell, a *vmspace* struktúrához hozzáadott változókat "példányosításkor" kinullázom.

```
diff --git a/sys/vm/vm_map.c b/sys/vm/vm_map.c
index 933b0e1..8754aa8 100644
--- a/sys/vm/vm_map.c
+++ b/sys/vm/vm_map.c
@@ -65,6 +65,8 @@
 #include <sys/cdefs.h>
```

```
 __FBSDID("$FreeBSD$");

+#include "opt_pax.h"
+
 #include <sys/param.h>
 #include <sys/systm.h>
 #include <sys/kernel.h>
@@ -297,6 +299,10 @@ vmspace_alloc(min, max)
         vm->vm_taddr = 0;
         vm->vm_daddr = 0;
         vm->vm_maxsaddr = 0;
+#ifdef PAX_ASLR
+       vm->vm_aslr_delta_mmap = 0;
+       vm->vm_aslr_delta_stack = 0;
+#endif /* PAX_ASLR */
         return (vm);
 }
```

3.2.2.3 vm_mmap

A *sys/vm/vm_mmap.c* file-ban található a *sys_mmap(...)* függvény, mely az *mmap(...)* rendszerhívást valósítja meg. Számunka ez azért fontos, mert ez a függvény felelős user space oldalról a memóriakezelésért. Amikor amd64 rendszeren dinamikusan foglalunk memóriát, akkor ez a függvény ad nekünk egy memória régiót.

A *sys_mmap(...)* függvényen belül hívom meg a *pax_aslr_mmap(...)* függvényt, mely a címen a randomizációt alkalmazza.

```
diff --git a/sys/vm/vm_mmap.c b/sys/vm/vm_mmap.c
index c17e9ce..9113fbb 100644
--- a/sys/vm/vm_mmap.c
+++ b/sys/vm/vm_mmap.c
@@ -45,6 +45,7 @@ __FBSDID("$FreeBSD$");

 #include "opt_compat.h"
 #include "opt_hwpmc_hooks.h"
+#include "opt_pax.h"

 #include <sys/param.h>
 #include <sys/systm.h>
@@ -89,6 +90,10 @@ __FBSDID("$FreeBSD$");
 #include <sys/pmckern.h>
 #endif

+#ifdef PAX_ASLR
+#include <sys/pax.h>
+#endif /* PAX_ASLR */
+
 int old_mlock = 0;
 SYSCTL_INT(_vm,   OID_AUTO,   old_mlock,   CTLFLAG_RW   |   CTLFLAG_TUN,
&old_mlock, 0,
     "Do not apply RLIMIT_MEMLOCK on mlockall");
@@ -197,6 +202,9 @@ sys_mmap(td, uap)
         struct file *fp;
         struct vnode *vp;
         vm_offset_t addr;
```

```
+#ifdef PAX_ASLR
+       vm_offset_t orig_addr;
+#endif /* PAX_ASLR */
        vm_size_t size, pageoff;
        vm_prot_t cap_maxprot, prot, maxprot;
        void *handle;
@@ -207,6 +215,9 @@ sys_mmap(td, uap)
        cap_rights_t rights;

        addr = (vm_offset_t) uap->addr;
+#ifdef PAX_ASLR
+       orig_addr = addr;
+#endif /* PAX_ASLR */
        size = uap->len;
        prot = uap->prot & VM_PROT_ALL;
        flags = uap->flags;
@@ -389,6 +400,9 @@ sys_mmap(td, uap)
 map:
        td->td_fpop = fp;
        maxprot &= cap_maxprot;
+#ifdef PAX_ASLR
+       pax_aslr_mmap(td, &addr, orig_addr, flags);
+#endif /* PAX_ASLR */
        error = vm_mmap(&vms->vm_map, &addr, size, prot, maxprot,
            flags, handle_type, handle, pos);
        td->td_fpop = NULL;
```

3.2.2.4 kern_exec

A *sys/kern/kern_exec.c* file-ban található módosítások több szempontból is fontosak. A process/thread létrehozásakor itt hívom meg az ASLR által használ változók inicializációját elvégző *pax_aslr_init(...)* függvényt. Ugyancsak ebben a szekcióban alkalmazzuk a stack-en a randomizációt a *pax_aslr_stack(...)* függvényhívással.

```
diff --git a/sys/kern/kern_exec.c b/sys/kern/kern_exec.c
index 3890157..1a3ede0 100644
--- a/sys/kern/kern_exec.c
+++ b/sys/kern/kern_exec.c
@@ -31,6 +31,7 @@ __FBSDID("$FreeBSD$");
 #include "opt_hwpmc_hooks.h"
 #include "opt_kdtrace.h"
 #include "opt_ktrace.h"
+#include "opt_pax.h"
 #include "opt_vm.h"

 #include <sys/param.h>
@@ -95,6 +96,10 @@ __FBSDID("$FreeBSD$");
 dtrace_execexit_func_t dtrace_fasttrap_exec;
 #endif

+#ifdef PAX_ASLR
+#include <sys/pax.h>
+#endif /* PAX_ASLR */
+
 SDT_PROVIDER_DECLARE(proc);
```

```
 SDT_PROBE_DEFINE(proc, kernel, , exec, exec);
 SDT_PROBE_ARGTYPE(proc, kernel, , exec, 0, "char *");
@@ -1044,6 +1049,10 @@ exec_new_vmspace(imgp, sv)
                map = &vmspace->vm_map;
        }

+#ifdef PAX_ASLR
+       pax_aslr_init(curthread, imgp);
+#endif /* PAX_ASLR */
+
        /* Map a shared page */
        obj = sv->sv_shared_page_obj;
        if (obj != NULL) {
@@ -1220,6 +1229,9 @@ exec_copyout_strings(imgp)
        int argc, envc;
        char **vectp;
        char *stringp, *destp;
+#ifdef PAX_ASLR
+       char *orig_destp;
+#endif /* PAX_ASLR */
        register_t *stack_base;
        struct ps_strings *arginfo;
        struct proc *p;
@@ -1248,6 +1260,10 @@ exec_copyout_strings(imgp)
            roundup(sizeof(canary), sizeof(char *)) -
            roundup(szps, sizeof(char *)) -
            roundup((ARG_MAX - imgp->args->stringspace), sizeof(char *));
+#ifdef PAX_ASLR
+       orig_destp = destp;
+       pax_aslr_stack(curthread, &destp, orig_destp);
+#endif /* PAX_ASLR */

        /*
         * install sigcode
```

3.2.2.5 kern_pax

A *sys/kern/kern_pax.c* file-ban találhatók a *sysctl handler*-ek – melyek segítségével futásidőben lehet változtatni az ASLR tulajdonságait - sokasága, a *tunables* definíciók, az ASLR által használt globális változók, és a randomizációt végző függvények a natív kódokhoz és az emulációs rétegekhez.

Első lépésben létrehozok egy saját node-ot a sysctl fában:

```
SYSCTL_NODE(_security, OID_AUTO, pax, CTLFLAG_RD, 0,
        "PaX (exploit mitigation) features.");
```

mely alá létrehozok még egy node-ot, amely az ASLR beállításait fogja tartalmazni:

```
SYSCTL_NODE(_security_pax, OID_AUTO, aslr, CTLFLAG_RD, 0,
        "Address Space Layout Randomization.");
```

(A választásom azért esett erre a kétszintű kezdeti felépítésre, mivel a közeljövőben foglalkozni szeretnék még ezzel a projekttel és az ASLR-en kívül több biztonsági funkciót is kívánok implementálni.)

A következő létrehozandó elem segítségével lehet állítani az ASLR hatáskörét. Mint ahogy az lejjebb látható, több szintet definiáltam.

0 érték esetén az ASLR ki van kapcsolva;

1 esetén az ASLR-t opcionálisan be lehet kapcsolni binárisonként;

2 esetén globálisan be van kapcsolva az ASLR, de binárisonként ki lehet kapcsolni;

3 esetén globálisan be van kapcsolva és nem lehet kikapcsolni.

Jelenleg implementálva a 0-ás és 3-as szint van. A köztes szintek implementálásához a rendszeren levő *ELF* file-okat ki kell egészíteni egy plusz szekcióval vagy note-tal, amely tartalmazza a megfelelő beállításokat. Ez *kernel linker* és *run-time linker* szinten is beavatkozást igényel. A módosításhoz szükséges kernel oldali kódok részben megvalósultak, de nincsenek bekapcsolva. A *kernel linker* oldal még megvalósításra vár.

```
SYSCTL_PROC(_security_pax_aslr, OID_AUTO, status,
        CTLTYPE_INT|CTLFLAG_RW|CTLFLAG_TUN,
        NULL, 0, sysctl_pax_aslr_status, "I",
        "Restrictions status. "
        "0 - disabled, "
        "1 - enabled,  "
        "2 - global enabled, "
        "3 - force global enabled");
TUNABLE_INT("security.pax.aslr.status", &pax_aslr_status);
```

Az ezt követő sysctl-ek és tunable-k az ASLR tulajdonságait állítják, hogy a randomizálandó címből hány bit legyen randomizálva. Egyet kiragadva közülük:

```
SYSCTL_PROC(_security_pax_aslr, OID_AUTO, mmap_len,
        CTLTYPE_INT|CTLFLAG_RW|CTLFLAG_TUN,
        NULL, 0, sysctl_pax_aslr_mmap, "I",
        "Number of bits randomized for mmap(2) calls. "
        "32 bit: [8,16] 64 bit: [16,32]");
TUNABLE_INT("security.pax.aslr.mmap", &pax_aslr_mmap_len);
```

És a hozzá tartozó sysctl handler:

```
static int
sysctl_pax_aslr_mmap(SYSCTL_HANDLER_ARGS)
{
        int     err;
```

```
        int     val;

        val = pax_aslr_mmap_len;
        err = sysctl_handle_int(oidp, &val, sizeof(int), req);
        if (err || !req->newptr)
                return (err);

        if (val < PAX_ASLR_DELTA_MMAP_MIN_LEN
            || val > PAX_ASLR_DELTA_MMAP_MAX_LEN)
                return (EINVAL);

        pax_aslr_mmap_len = val;

        return (0);
}
```

A lényegi megvalósítás, amelyek az ASLR-t implementálják, a *pax_aslr_init(...)* függvényben és az ezt követőekben találhatók:

```
void
pax_aslr_init(struct thread *td, struct image_params *imgp)
{
        struct vmspace *vm;
        u_int sv_flags;

        if (imgp == NULL) {
                panic("[PaX ASLR] pax_aslr_init - imgp == NULL");
        }

        if (!pax_aslr_active(td))
                return;

        vm = imgp->proc->p_vmspace;
        sv_flags = imgp->proc->p_sysent->sv_flags;
#ifndef COMPAT_FREEBSD32
        vm->vm_aslr_delta_mmap = PAX_ASLR_DELTA(arc4random(),
                PAX_ASLR_DELTA_MMAP_LSB, pax_aslr_mmap_len);
        vm->vm_aslr_delta_stack = PAX_ASLR_DELTA(arc4random(),
                PAX_ASLR_DELTA_STACK_LSB, pax_aslr_stack_len);
        vm->vm_aslr_delta_stack = ALIGN(vm->vm_aslr_delta_stack);
#else /* COMPAT_FREEBSD32 */
        if ((sv_flags & SV_LP64) != 0) {
                vm->vm_aslr_delta_mmap = PAX_ASLR_DELTA(arc4random(),
                        PAX_ASLR_DELTA_MMAP_LSB, pax_aslr_mmap_len);
                vm->vm_aslr_delta_stack = PAX_ASLR_DELTA(arc4random(),
                        PAX_ASLR_DELTA_STACK_LSB, pax_aslr_stack_len);
                vm->vm_aslr_delta_stack = ALIGN(vm->vm_aslr_delta_stack);
        } else {
                vm->vm_aslr_delta_mmap = PAX_ASLR_DELTA(arc4random(),
                        PAX_ASLR_COMPAT_DELTA_MMAP_LSB,
pax_aslr_compat_mmap_len);
                vm->vm_aslr_delta_stack = PAX_ASLR_DELTA(arc4random(),
                        PAX_ASLR_COMPAT_DELTA_STACK_LSB,
pax_aslr_compat_stack_len);
                vm->vm_aslr_delta_stack = ALIGN(vm->vm_aslr_delta_stack);
        }
#endif /* !COMPAT_FREEBSD32 */
}
```

Ez a függvény minden thread létrehozásakor lefut és inicializálja az ASLR által használt változókat, melyek alapján megtörténik a randomizáció. A FreeBSD kernelt amd64 architektúrán lehet fordítani kompatibilitási réteggel és anélkül. Erre az esetre az init függvény kvázi két részre van osztva sebesség optimalizálás miatt. Ha nincs kompatibilitási réteg, akkor csak a natív rész inicializálódik, ha van, akkor a kód típusának megfelelő.

Az *mmap(...)* randomizációt a *pax_aslr_mmap(...)* függvény végzi, a *pax_aslt_init(...)* által generált random alapján. A randomizációt csak bizonyos esetekben alkalmazzuk. Az egyik ilyen eset, ha nem *MAP_FIXED* flag-gel hívjuk meg az *mmap(...)*-ot és *NULL* kezdőcímű hintet adunk neki (azaz a kernelre bízzuk a cím kiválasztását), ha *NULL*-tól eltérő, akkor nem randomizáljuk, mivel ez részben törné az *mmap(...)* alap funkcionalitását. A másik ilyen eset, ha mappelés nem *MAP_ANON*, azaz nem tartozik hozzá semmilyen file descriptor, hanem csak memória régiót kérünk a kerneltől, ez utóbbit implikálja a *MAP_STACK* is.

```
void
pax_aslr_mmap(struct thread *td, vm_offset_t *addr, vm_offset_t orig_addr,
int flags)
{
        if (!pax_aslr_active(td))
                return;

        if (!(flags & MAP_FIXED) && ((orig_addr == 0) || !(flags &
MAP_ANON))) {
#ifdef PAX_ASLR_DEBUG
                uprintf("[PaX ASLR] applying to %p orig_addr=%p f=%x\n",
                    (void *)*addr, (void *)orig_addr, flags);
#endif /* PAX_ASLR_DEBUG */
                if        (!(td->td_proc->p_vmspace->vm_map.flags        &
MAP_ENTRY_GROWS_DOWN))
                        *addr          +=          td->td_proc->p_vmspace-
>vm_aslr_delta_mmap;
                else
                        *addr          -=          td->td_proc->p_vmspace-
>vm_aslr_delta_mmap;
#ifdef PAX_ASLR_DEBUG
                uprintf("[PaX ASLR] result %p\n", (void *)*addr);
#endif /* PAX_ASLR_DEBUG */
        }
#ifdef PAX_ASLR_DEBUG
        else
            uprintf("[PaX ASLR] not applying to %p orig_addr=%p f=%x\n",
                (void *)*addr, (void *)orig_addr, flags);
#endif /* PAX_ASLR_DEBUG */
}
```

A stack randomizáció esetén jelenleg stack gap randomizáció van megvalósítva, ami annyit jelent, hogy a lapon belül van eltolva a stack kezdőcíme:

```
void
pax_aslr_stack(struct thread *td, char **addr, char *orig_addr)
{
        if (!pax_aslr_active(td))
                return;

        *addr -= td->td_proc->p_vmspace->vm_aslr_delta_stack;
#ifdef PAX_ASLR_DEBUG
        uprintf("[PaX ASLR] orig_addr=%p, addr=%p\n",
            (void *)orig_addr, (void *)*addr);
#endif /* PAX_ASLR_DEBUG */
}
```

A stack randomizációt a *sys/kern/kern_exec.c* file-ban található *exec_copyout_strings(...)* függvényben hívom meg, mely összeállítja a program futtatás linkerhez szükséges adatstruktúrát és beállítja a user space stack kezdőcímét.

3.2.2.6 Módosítások összegzése

```
git                                diff                                --stat
94108ee710c3afd3c9944ed40e129598dd242a00..54316e66bd3e82d9b02721bbc2c7c8a3
f7254e81
 sys/compat/freebsd32/freebsd32_misc.c |  13 +
 sys/conf/files                        |   1 +
 sys/conf/options                      |   5 +
 sys/kern/kern_exec.c                  |  16 ++
 sys/kern/kern_pax.c                   | 450 ++++++++++++++++++++++++++++++++++
 sys/sys/pax.h                         | 161 ++++++++++++
 sys/vm/vm_map.c                       |   6 +
 sys/vm/vm_map.h                       |   2 +
 sys/vm/vm_mmap.c                      |  14 ++
 9 files changed, 668 insertions(+)
```

3.2.3 Kezdeti beállítások

A kezdeti beállításokat az adott architektúra szóhossza szerint állítom be. A randomizációk alsó és felső korlátait a következő képletek adják meg:

	min	***max***
mmap	`((sizeof(void *) * NBBY) / 4)`	`((sizeof(void *) * NBBY) / 2)`
stack	`((sizeof(void *) * NBBY) / 5)`	`((sizeof(void *) * NBBY) / 3)`
exec_base	`((sizeof(void *) * NBBY) / 5)`	`((sizeof(void *) * NBBY) / 3)`

Kompatibilitási esetben a képletek hasonlóan alakulnak, azonban egy fontos különbség van: *void* * pointer mérete helyett az *int* mérete alapján számolom ki a randomizáció hosszát.

	min	*max*
mmap	`((sizeof(int) * NBBY) / 4)`	`((sizeof(int) * NBBY) / 2)`
stack	`((sizeof(int) * NBBY) / 5)`	`((sizeof(int) * NBBY) / 3)`
exec_base	`((sizeof(int) * NBBY) / 5)`	`((sizeof(int) * NBBY) / 3)`

Az értékek alakulása: 64 bites rendszeren a *sizeof(void *)* értéke *8*, 32 bites rendszeren pedig *4*; az *NBBY* egy FreeBSD által architektúránként definiált makró, amely megadja, hogy az adott CPU esetében egy byte hány bitet tartalmaz. Az osztók pedig súlyok, amelyek megadnak egy elégséges alsó határt és egy felső határt, mely esetében már hibák is léphetnek fel.

Kompatibilitási esetben a döntés azért esett az *int* típusra, mert a FreeBSD által támogatott architektúrákon az *int* rendre *32 bit* hosszúként van értelmezve, így a *sizeof(int)* mindig *4*-et fog visszaadni és nekünk pont ez kell.

Az konkrét értékek pedig:

Számítási mód	*érték*
`((sizeof(void *) * NBBY) / 2)`	32
`((sizeof(void *) * NBBY) / 3)`	21
`((sizeof(void *) * NBBY) / 4)`	16
`((sizeof(void *) * NBBY) / 5)`	12
`((sizeof(int) * NBBY) / 2)`	16
`((sizeof(int) * NBBY) / 3)`	10
`((sizeof(int) * NBBY) / 4)`	8
`((sizeof(int) * NBBY) / 5)`	6

Alapértelmezetten a globális ASLR beállítások a minimumra vannak állítva, mivel így a legkisebb a valószínűsége, hogy memóriaigényes programok esetén hiba lépne fel, ezt a tulajdonságot több módon is megváltoztathatjuk. Az egyik megoldás, ha a FreeBSD rendszerekben található */etc/sysctl.conf* file segítségével átállítjuk a kívánt értékre a sysctl változókat. A másik megoldás, ha a */boot/loader.conf* file-ba írjuk bele a megfelelő beállításokat, amelyeket aztán a *tunable* alrendszer boot időben beállít. Végül a harmadik módszer, hogy fordítás időben állítjuk a legbiztonságosabbra, melyet a *PAX_ASLR_MAX_SEC* opcióval lehet beállítani.

3.2.4 Buktatók

Ha az értékeket túl magasra állítjuk, akkor bizonyos programok nem indulnak el rendesen, ha túl alacsonyra, akkor pedig sérülékenyek lehetnek a támadásokra nézve.

Az ASLR önmagában nem megoldás / workaround, mivel több módszerrel is támadható[17][31], azonban még így is egy hatékony eszköz a védelem oldaláról. Bizonyos sérülékenységeit hatékonyan lehet orvosolni a segvguard megoldással, melyről a bevezetőben volt szó.

3.2.5 Emulációs rétegek

3.2.5.1 linuxulator

A *linuxulator* a FreeBSD egyik hibrid megoldása, mely 32 bites Linuxra elkészített ELF binárisok futtatását teszi lehetővé. A probléma ezzel az, hogy nem tisztán van megvalósítva. A *linuxulator* emulációs réteg a Linux által használt rendszerhívások sokaságát fordítja át natív FreeBSD rendszerhívásokra. A gond ezzel ott van, hogy amd64 rendszeren nem a *freebsd32* emulációs rétegre fordítja át, hanem a natív 64 bites rendszerhívásokra. Így, ha a rendszeren Linuxra készült binárisokat akarunk futtatni, akkor nem használhatjuk ki teljesen az ASLR adta lehetőségeket, mert az adott alkalmazás nem fog elindulni.

3.2.5.2 freebsd32

FreeBSD amd64 alatt lehetőség van 32 bites FreeBSD rendszerekhez készült ELF binárisok futtatására is, mely a *linuxulator*-hoz hasonlóan natív függvényhívásokat eredményez azzal a különbséggel, hogy a *freebsd32* emulációs rétegben – a linuxulator-ral szemben – nem lépnek fel problémák. A rendszerben különböző esetekben felmerülő problémák kezelve vannak. Ez érthető is, mivel korábbi FreeBSD verziók binárisainak futtatására nagyobb igény van, mint más rendszer binárisainak futtatására.

Ebből kifolyólag az ASLR megfelelő inicializációját meg tudtam oldani freebsd32 emulációs réteghez is.

4 Értékelés

4.1 SMAP

Az technológia beváltotta a hozzá fűzött reményeket, így a nem engedélyezett user space memória hozzáférés esetén a kernel *panic(...)* függvényhívásba végződik. Ez biztonsági szempontból fontos előrelépés, mivel így a támadó nem képes hozzáférni érzékeny adatokhoz. A technológia hátránya, hogy a gép ezt követően használhatatlan lesz a következő újraindításig.

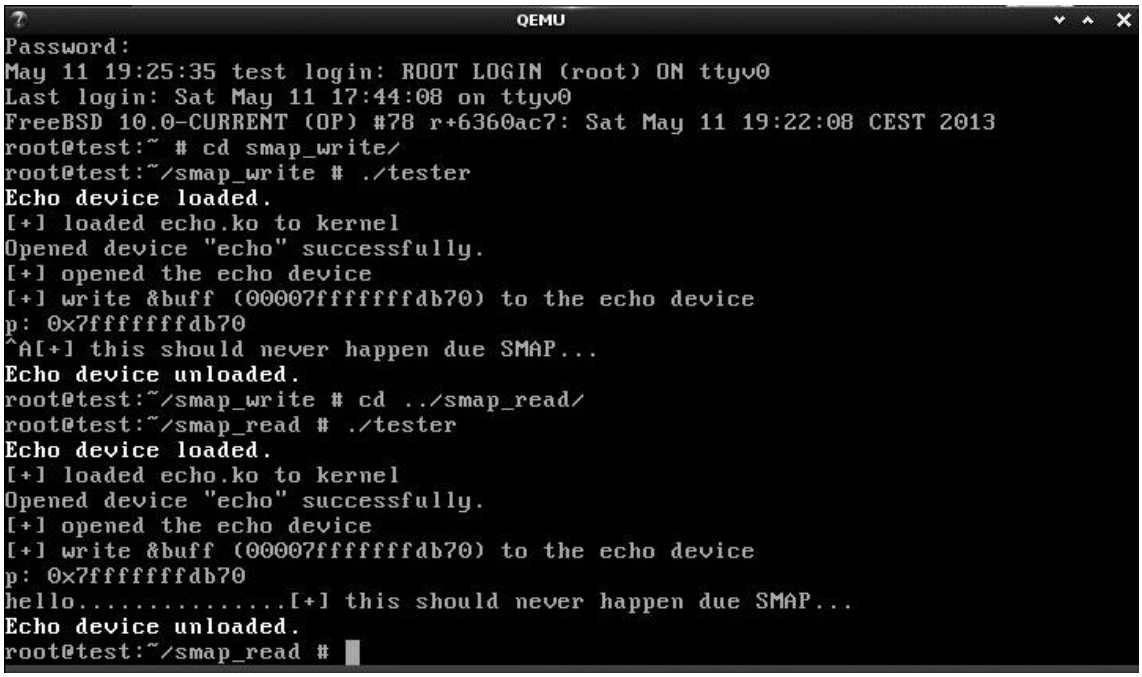


Figure 6: user space memória írása, olvasása kernelből SMAP nélkül

Mint ahogy az a (Figure 6: user space memória írása, olvasása kernelből SMAP nélkül) ábrán látható, ha az SMAP védelem nincs bekapcsolva, akkor a kernel képes nem kontrollált módon is user space memóriát olvasni. Ennek a következménye súlyos lehet egy esetleges sérülékenység kihasználásánál, mivel a kernelbe nem kernel kód injektálható.

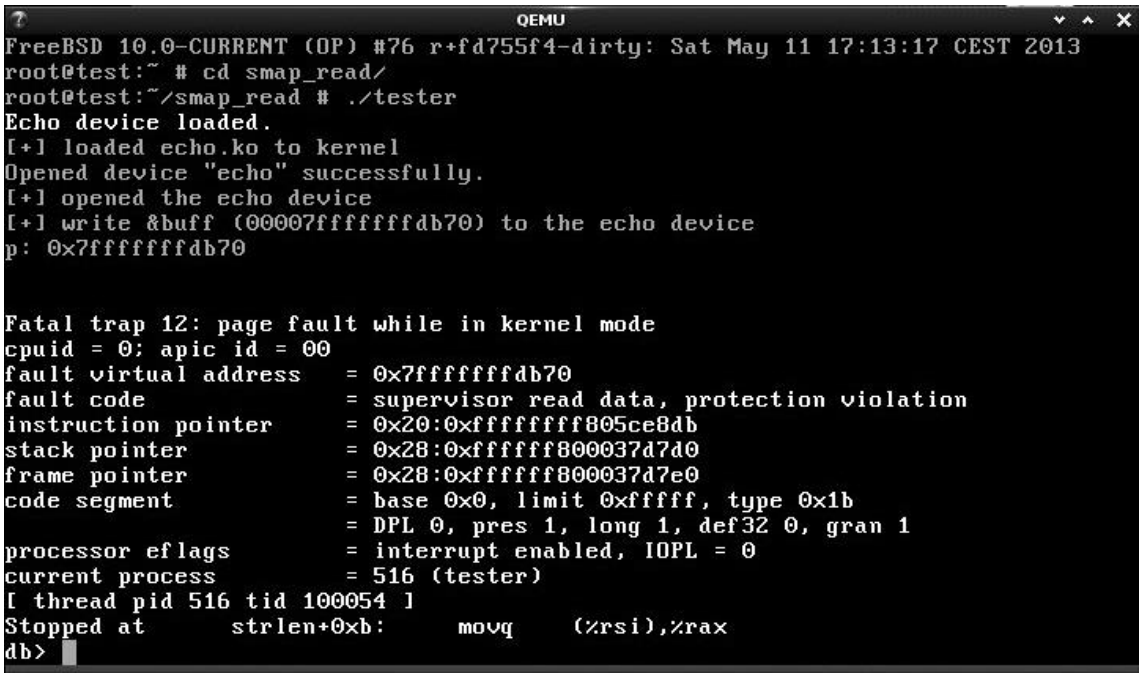


Figure 7: user space memória olvasása bekapcsolt SMAP védelemmel

Figure 8: user space memória írása bekapcsolt SMAP védelemmel

Ha az SMAP védelem be van kapcsolva, akkor az ahogy a Figure 7 ábrán látható - olvasás esetén és írás esetén - a Figure 8 ábrán látható - kernel *panic(...)*-ot kapunk, ezáltal megakadályozva a kódinjektálást.

A technológia pozitív mellékhatása, hogy a kernel kód minősége javulhat, mivel a nem megfelelő memória hozzáférés esetén a kernel nem lesz funkcionális és ki kell javítani a programozói hibákat.

Az elkészült módosításról teljesítmény tesztek egyelőre nem készültek, mivel olyan fizikai géphez még nem fértem hozzá, amely támogatja az SMAP technológiát. A tesztek Qemu 1.4 alatt készültek.

4.2 ASLR

Az ASLR egy kis költségű, de hatékony technika, mely megnehezíti a rendszerek exploitálását[35].

FreeBSD alatt az implementációt hatékonyan meg lehetett oldani, mivel rendelkezésre álltak a szükséges framework-ök, mint például a sysctl, tunables, *arc4random(...)*, stb.

4.2.1 ASLR tesztesetek

4.2.1.1 A memory mapping ellenőrzése procstat-tal

A következő sorok leírják, hogy az egyes alrendszerek randomizációjakor mennyi bitet fogunk felhasználni (megjegyzendő itt is, hogy az exec base, még nincs implementálva, ugyancsak implementációs részben kifejtve).

Első lépés a jelenlegi beállítások ellenőrzése:

```
op@pandora-test ~> sysctl security.pax.aslr.
security.pax.aslr.status: 1
security.pax.aslr.mmap_len: 16
security.pax.aslr.stack_len: 12
security.pax.aslr.exec_len: 12
security.pax.aslr.compat.status: 1
security.pax.aslr.compat.mmap_len: 8
security.pax.aslr.compat.stack_len: 6
security.pax.aslr.compat.exec_len: 6
```

Az ASLR be van kapcsolva, mivel a *security.pax.aslr.status* értéke *1* (implementálva jelenleg a *0* és *3* státusz van jelenleg, de ami *0*-tól eltérő, az *3*-as szintként van értelmezve, mint ahogy azt az implementációs részben ki is fejtettem).

A *security.pax.aslr.compat* sysctl node megléte miatt látható, hogy a rendszer - melyen a teszteket végeztem - *freebsd32* kompatibilitási réteggel lett fordítva.

A következőkben a FreeBSD rendszer részét képező *cat* program address space-ét fogom vizsgálni a *procstat[36]* programmal.

A *cat* programot elindítom a háttérben, és a shell kiírja a hozzá tartozó PID-et. A *procstat[36] man* (a *man* parancs segítségével érjük el a manuálokat) oldalából erre van most szükségem:

```
-v      Display virtual memory mappings for the process.
```

Ez alapján a *procstat -v* parancs segítségével ki tudom íratni a process által mappelt memória régiókat. (Ez többek között potenciális infoleak).

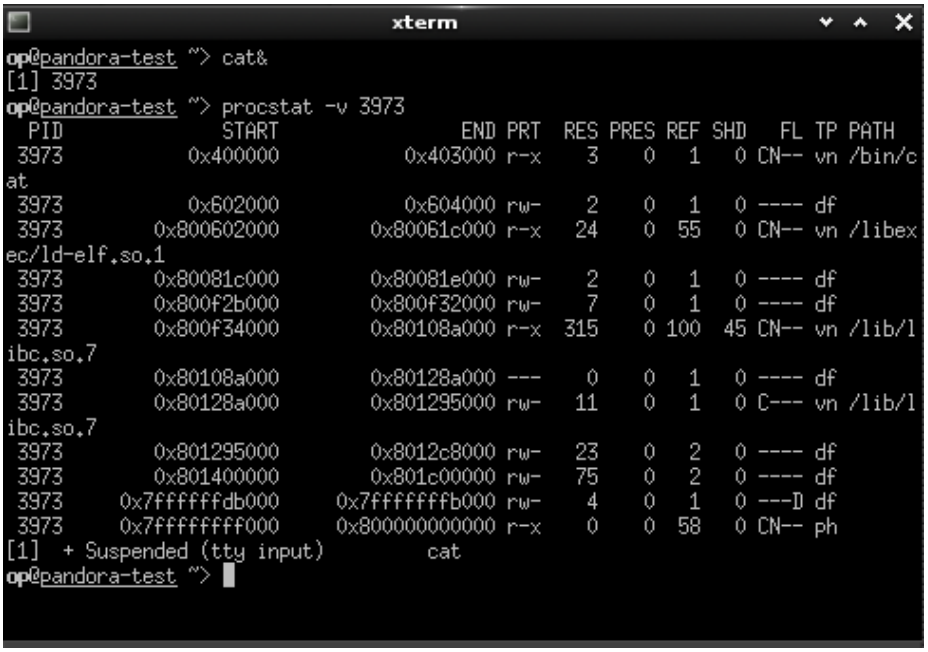


Figure 9: cat memory mapping ASLR-rel

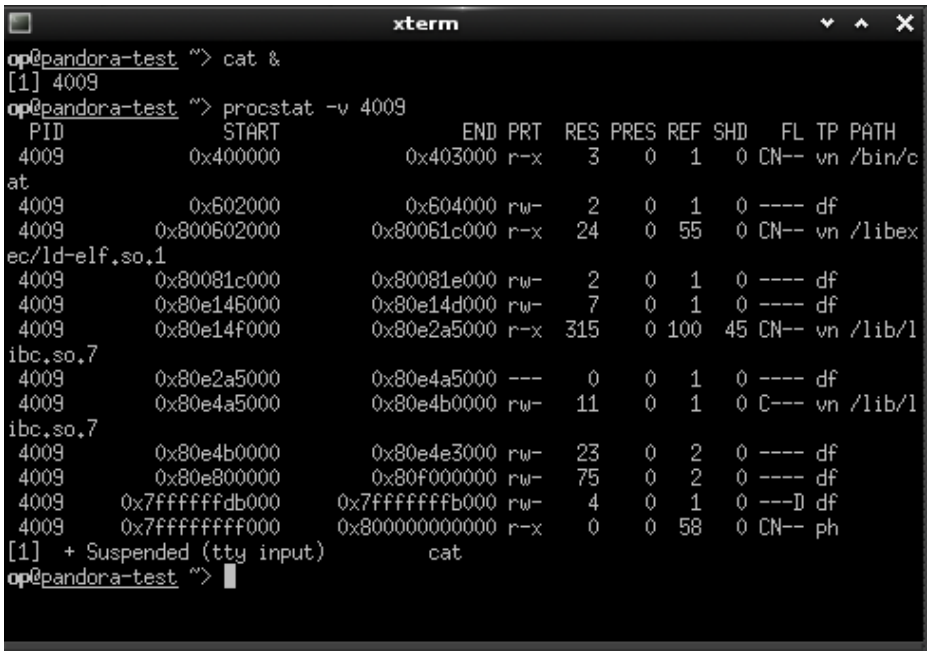


Figure 10: cat memory mapping ASLR-rel a következő futáskor

Látható a Figure 9 és Figure 10 ábrán, hogy két futás között a memória címek változtak. Ez várható volt, mivel ez az ASLR lényege, azonban ahogy az feltűnhet, a program belépési pontja és a linker nem lett randomizálva.

A heap randomizálva lett, a betöltött megosztott függvénykönyvtárak és a szegmensei is randomizálva lettek.

Az utolsó előtti memória cím a stacket jelenti. Erről a képről nem látszik, de az is randomizálva lett a stack gap megoldással, amit a későbbiekben igazolni is fogok.

```
op@pandora-test ~> su
Password:
op@pandora-test op# sysctl security.pax.aslr.status=0
security.pax.aslr.status: 1 -> 0
op@pandora-test op# exit
```

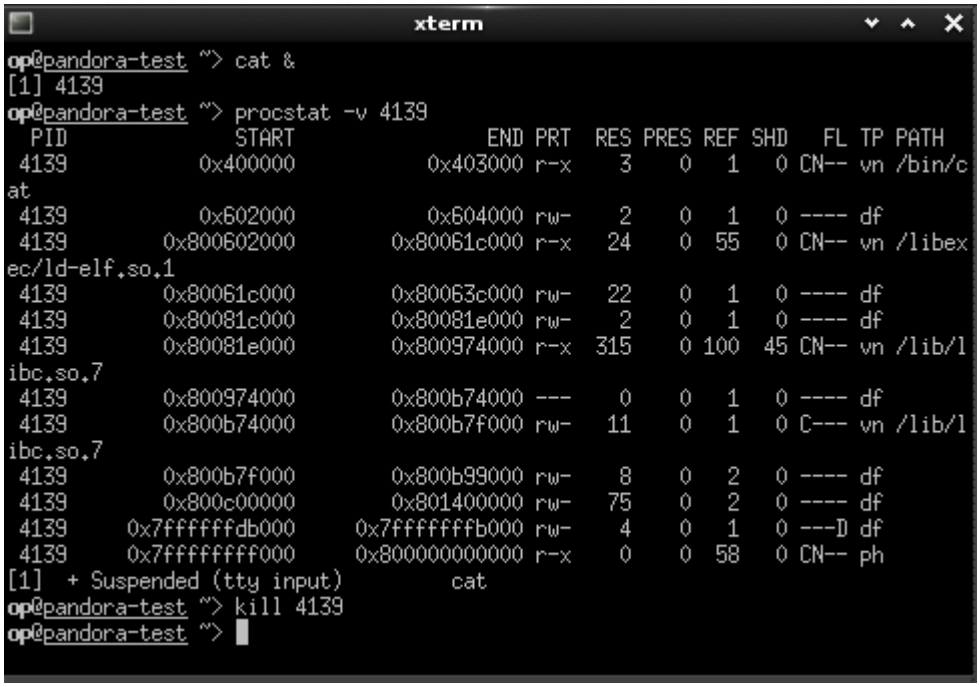


Figure 11: cat memory mapping ASLR nélkül

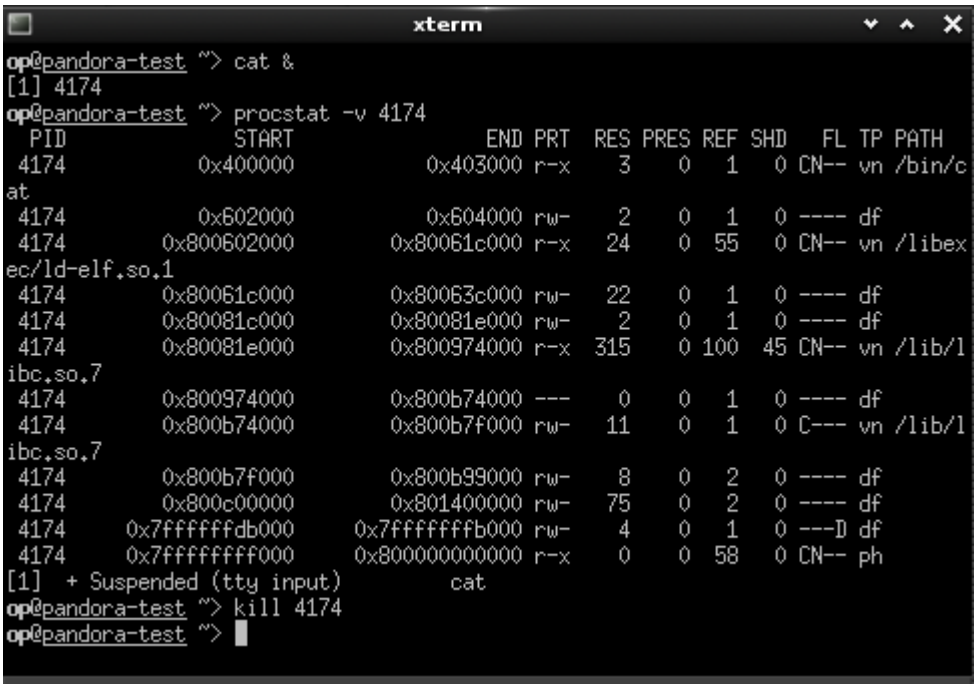


Figure 12: cat memory mapping ASLR nélkül a következő futtatáskor

A Figure 11 és Figure 12 ábrát megelőzendő parancsok segítségével kikapcsoltam az ASLR-t és megismételtem a fent leírtakat.

Ebben az esetben a két futás között semmilyen különbség nem tapasztalható a megváltozott PID-eken kívül.

4.2.1.2 ASLR randomizáció ellenőrzése paxtest-tel

A következő eszköz, amely segítségével az implementációt vizsgáltam a *paxtest*[37], melyet több helyen is módosítanom kellett, hogy fordítható és futtatható legyen FreeBSD alatt is: ebből lett a *paxtest-freebsd*[38]. A *paxtest* egy egész sor ellenőrzést végez, különböző szegmensek írhatóságával és futtathatóságával, a randomizációjával és egyéb ellenőrzésekkel.

Az ASLR viszonylatában most engem a randomizációt érintő tesztek érdekeltek.

A program kimenete bekapcsolt ASLR esetén a következő:

```
op@pandora-test paxtest-freebsd# sysctl security.pax.aslr.status=1
security.pax.aslr.status: 0 -> 1

op@pandora-test paxtest-freebsd> ./paxtest kiddie
PaXtest - Copyright(c) 2003,2004 by Peter Busser <peter@adamantix.org>
Released under the GNU Public Licence version 2 or later

Writing output to paxtest.log
It may take a while for the tests to complete
Test results:
```

```
PaXtest - Copyright(c) 2003,2004 by Peter Busser <peter@adamantix.org>
Released under the GNU Public Licence version 2 or later

Mode: kiddie
FreeBSD pandora-test 10.0-CURRENT FreeBSD 10.0-CURRENT #80 r+70a9d86: Mon
May     13     16:48:55     CEST     2013                       op@pandora-
test:/usr/obj/usr/home/op/git/freebsd-base.git.http/sys/OP  amd64

Executable anonymous mapping             : Killed
Executable bss                           : Killed
Executable data                          : Killed
Executable heap                          : Killed
Executable stack                         : Killed
Executable anonymous mapping (mprotect)  : Vulnerable
Executable bss (mprotect)                : Vulnerable
Executable data (mprotect)               : Vulnerable
Executable heap (mprotect)               : Vulnerable
Executable shared library bss (mprotect) : Vulnerable
Executable shared library data (mprotect): Vulnerable
Executable stack (mprotect)              : Vulnerable
Anonymous mapping randomisation test     : 16 bits (guessed)
Heap randomisation test (ET_EXEC)        : 7 bits (guessed)
Main executable randomisation (ET_EXEC)  : No randomisation
Shared library randomisation test        : 16 bits (guessed)
Stack randomisation test (SEGMEXEC)      : 12 bits (guessed)
Stack randomisation test (PAGEEXEC)      : 12 bits (guessed)
Arg/env randomisation test (SEGMEXEC)    : 13 bits (guessed)
Arg/env randomisation test (PAGEEXEC)    : 13 bits (guessed)
Return to function (strcpy)              : paxtest: return address contains
a NULL byte.
Return to function (strcpy, PIE)         : paxtest: return address contains
a NULL byte.
Return to function (memcpy)              : Vulnerable
Return to function (memcpy, PIE)         : Vulnerable
Executable shared library bss            : Killed
Executable shared library data           : Killed
Writable text segments                   : Vulnerable
```

A vastaggal kiemelt részen látható a program által valószínűsített randomizáció, mely közel egybecseng a beállított értékekkel. Az mmap randomizáció esetén 16 bit-es beállítások voltak, a stack randomizáció esetén pedig 12 bit-es. Ezek szépen látszanak a fenti kimenetből is.

A tesztet megismételtem kikapcsolt ASLR-rel is, melynek a kimenete ez:

```
op@pandora-test paxtest-freebsd> su
Password:
op@pandora-test paxtest-freebsd# sysctl security.pax.aslr.status=0
security.pax.aslr.status: 1 -> 0
op@pandora-test paxtest-freebsd# exit

op@pandora-test paxtest-freebsd> ./paxtest kiddie
PaXtest - Copyright(c) 2003,2004 by Peter Busser <peter@adamantix.org>
Released under the GNU Public Licence version 2 or later

Writing output to paxtest.log
```

```
It may take a while for the tests to complete
Test results:
PaXtest - Copyright(c) 2003,2004 by Peter Busser <peter@adamantix.org>
Released under the GNU Public Licence version 2 or later

Mode: kiddie
FreeBSD pandora-test 10.0-CURRENT FreeBSD 10.0-CURRENT #80 r+70a9d86: Mon
May     13     16:48:55     CEST     2013                       op@pandora-
test:/usr/obj/usr/home/op/git/freebsd-base.git.http/sys/OP  amd64

Executable anonymous mapping             : Killed
Executable bss                           : Killed
Executable data                          : Killed
Executable heap                          : Killed
Executable stack                         : Killed
Executable anonymous mapping (mprotect)  : Vulnerable
Executable bss (mprotect)                : Vulnerable
Executable data (mprotect)               : Vulnerable
Executable heap (mprotect)               : Vulnerable
Executable shared library bss (mprotect) : Vulnerable
Executable shared library data (mprotect): Vulnerable
Executable stack (mprotect)              : Vulnerable
Anonymous mapping randomisation test     : No randomisation
Heap randomisation test (ET_EXEC)        : No randomisation
Main executable randomisation (ET_EXEC)  : No randomisation
Shared library randomisation test        : No randomisation
Stack randomisation test (SEGMEXEC)      : No randomisation
Stack randomisation test (PAGEEXEC)      : No randomisation
Arg/env randomisation test (SEGMEXEC)    : No randomisation
Arg/env randomisation test (PAGEEXEC)    : No randomisation
Return to function (strcpy)              : paxtest: return address contains
a NULL byte.
Return to function (strcpy, PIE)         : paxtest: return address contains
a NULL byte.
Return to function (memcpy)              : Vulnerable
Return to function (memcpy, PIE)         : Vulnerable
Executable shared library bss            : Killed
Executable shared library data           : Killed
Writable text segments                   : Vulnerable
```

Ugyancsak a vastaggal kiemelt részeket érdemes figyelni. Ebben az esetben semmilyen randomizáció nem történik, ahogy azt el is vártuk.

A jelenleg elkészített megoldás még nem tökéletes és csak részben implementálja a PaX Team által leírt megoldásokat, de az implementáció jelenlegi állapotával szemben támasztott követelményeimet teljesíti.

A projektet egyetemen kívül is folytatni szeretném, mert napi szinten használok FreeBSD-t.

4.2.1.3 ASLR implementáció valós körülmények között

A patchet különböző fázisaiban felraktam a saját gépemre, melyen a fejlesztések készültek. Ez idő alatt rendellenességet nem tapasztaltam.

A tipikus használat az alábbiakból állt:

- Firefox és Opera böngésző használata több tab-bal;
- Vim editor clang_complete plugin-nel több példányban használata;
- FreeBSD build rendszer használata nagyobb módosítások után, ez közel 1.5 órás intenzív load, mely jelentős számítási, disc műveletet eredményez, ezen felül számos process és thread kerül létrehozásra;
- FreeBSD build rendszer használata kisebb módosítások után;
- linuxulator-ral Linuxra készült alkalmazások futtatása;
- *vlc*-vel HD videó nézése, *mocp*-vel zenehallgatás;
- proxyzás és filterezés;
- Qemu alatt FreeBSD futtatása.

Ezen fentebb felsorolt pontokat egyszerre párhuzamosan futattam.

A gép egy 4 magos 64-bites Intel CPU-val rendelkezik, melynek típusa Q9300, a gép 4 GB fizikai memóriát tartalmaz, az alaplap Asus P5Q-E és a rendszer UFS file rendszeren van. Ezek az információk a következő pontban ismertetett *unixbench* esetében is relevánsak.

4.2.1.4 ASLR implemenáció teljesítmény tesztje unixbench programmal

A teljesítmény tesztekhez a *unixbench[39]* nevű tesztet használtam, abból is a 4.1-es verziót.

A patch és a teszt amd64 architektúrájú FreeBSD 10-CURRENT alatt készült, r249952 revision alatt. A rendszer single user módban lett elindítva, ezt követően a filerendszerek fel lettek csatolva a *mount -a* parancs segítségével. A tesztek *script* parancs alatt futottak, melynek célja, hogy minden terminálra kerülő outputot és inputot eltárol egy file-ban.

	disabled aslr		*enabled aslr*		*without aslr*		
	avg	szoras	**avg**	szoras	**avg**	szoras	
C Compiler Throughput	**1089.866667**	2.615976554	**1093.3**	3.750999867	**1092.5**	3.174901573	lpm
Dc: sqrt(2) to 99 decimal places	**83885.93333**	214.9895424	**83427**	786.7425691	**83637.5**	104.4307905	lpm
Dhrystone 2 using register variables	**19864723.33**	23882.548	**19698426.2**	136015.0205	**19228373.7**	202126.5748	lps
Double-Precision Whetstone	**3259.933333**	1.955334583	**3245.466667**	21.86603149	**3261.033333**	2.638812864	MWIPS
Execl Throughput	**1262.533333**	7.1988425	**1260.733333**	4.630694692	**1270.666667**	4.717343885	lps
File Copy 1024 bufsize 2000 maxblocks	**93897**	1569.701883	**94943.33333**	192.2246949	**94597.66667**	434.1432175	KBps
File Copy 256 bufsize 500 maxblocks	**26269.66667**	647.5216856	**26089**	454.808751	**26073**	467.9882477	KBps
File Copy 4096 bufsize 8000 maxblocks	**73755**	77.48548251	**72271.33333**	1255.538663	**73772.66667**	28.14841618	KBps
File Read 1024 bufsize 2000 maxblocks	**281040.3333**	2438.549227	**280970.6667**	1000.484549	**273580**	193.1864384	KBps
File Read 256 bufsize 500 maxblocks	**71493.33333**	847.0385666	**71552**	435.1689327	**70263.66667**	106.2277428	KBps
File Read 4096 bufsize 8000 maxblocks	**912792**	3980.592795	**907621.3333**	3458.126998	**886477**	2924.608863	KBps
File Write 1024 bufsize 2000 maxblocks	**151343.3333**	420.9089371	**154922**	3107.584914	**155800**	3335.458139	KBps
File Write 256 bufsize 500 maxblocks	**45039**	670.5005593	**45024**	783.171118	**45834.66667**	738.9061736	KBps
File Write 4096 bufsize 8000 maxblocks	**68977.33333**	88.50047081	**67631**	1165.670193	**68333.66667**	1192.193077	KBps
Pipe Throughput	**685209.3**	3467.610213	**685788.1**	3643.923557	**672573.9667**	1877.654426	lps
Pipe-based Context Switching	**120032.7**	2072.9011	**122654.7667**	1292.200764	**119311.7667**	2337.814044	lps
Process Creation	**3032.666667**	55.55558778	**3070.966667**	77.27873791	**3050.4**	61.51349445	lps
Recursion Test--Tower of Hanoi	**189751.0333**	264.2940849	**189276.4**	725.3452419	**190272.0667**	416.1870052	lps
Shell Scripts (1 concurrent)	**3707.966667**	37.44520441	**3690.266667**	47.68021952	**3727.3**	40.42635279	lpm
Shell Scripts (16 concurrent)	**490.9666667**	1.517673658	**487.2**	3.551056181	**493.1333333**	1.4571662	lpm
Shell Scripts (8 concurrent)	**941.4333333**	8.256108849	**938**	11.17273467	**947.7666667**	11.24825912	lpm
System Call Overhead	**536400.5333**	912.0273369	**534672.8**	968.5840593	**537497.4667**	1931.36515	lps

Figure 13: unixbench program futásából kapott végeredmények

A fenti eredményeket a *unixbench* program háromszori futtatásával kaptam. A program a teszteseteit háromszor ismétli meg, így összesen minden teszteset 9 alkalommal futott le. A fenti táblázat ezen értéken átlagát és szórását tartalmazza. Az utolsó oszlopban pedig a dimenziók találhatók. Jelentésük: lpm – loop per minute, MWIPS - Millions of Whetstone Instructions Per Second, lps – loop per sec.

Az elkövetkező oldalakon pár fontosabb mérési eredményt vizuálisan is megmutatok.

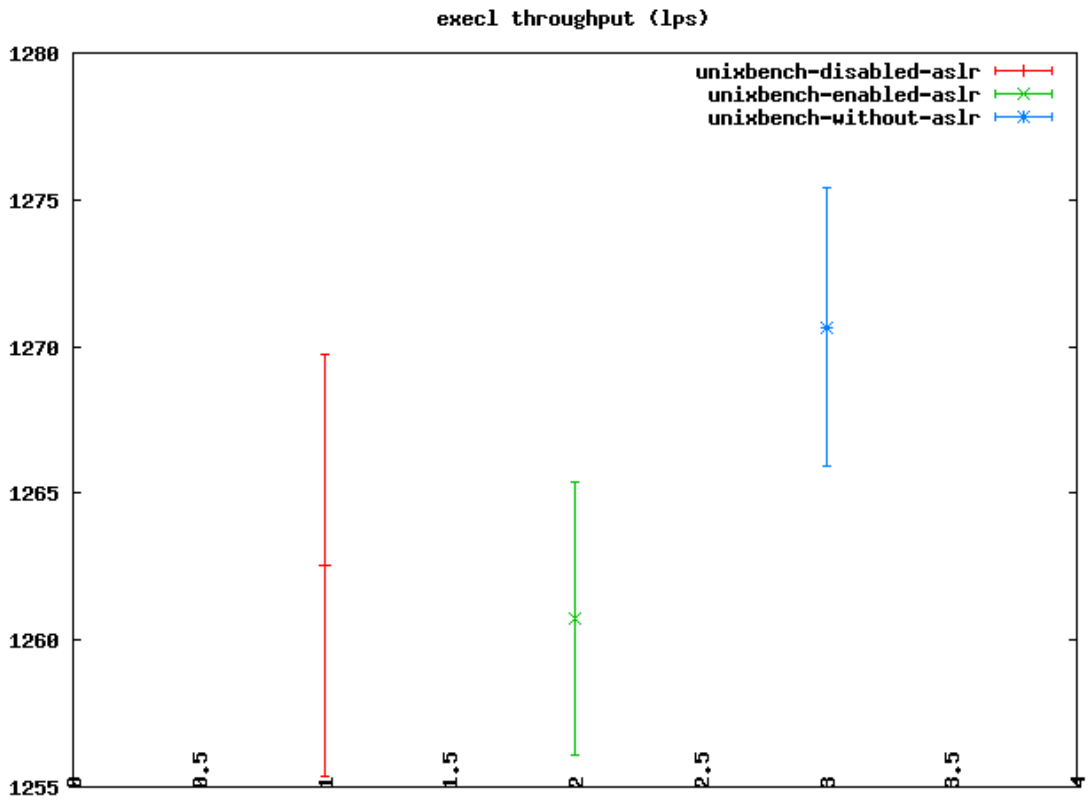


Figure 14: execl throughput (loops per sec)

A Figure 14: execl throughput (loops per sec) ábrán a másodpercenként lefutott *execl(...)* függvényhívás számát láthatjuk. Az *execl(...)* az aktuális process image-t cseréli le az újra.

Az előzetes várakozásaimnak megfelelően ebben a tesztesetben a patch nélküli kernel volt a leggyorsabb. Ezt a megpatch-elt, de kikapcsolt ASLR-rel használt kernel, míg a leglassabb az ASLR-rel használt kernel volt.

A jelenség eredete, hogy bekapcsolt ASLR esetében a kernel legenerálja a szükséges randomokat, és ellenőrzi a feltételeket, hogy melyik ASLR beállítás érvényes az adott process-re.

Kikapcsolt ASLR esetében a randomok nem generálódnak le, csak az ellenőrzés költsége tevődik hozzá.

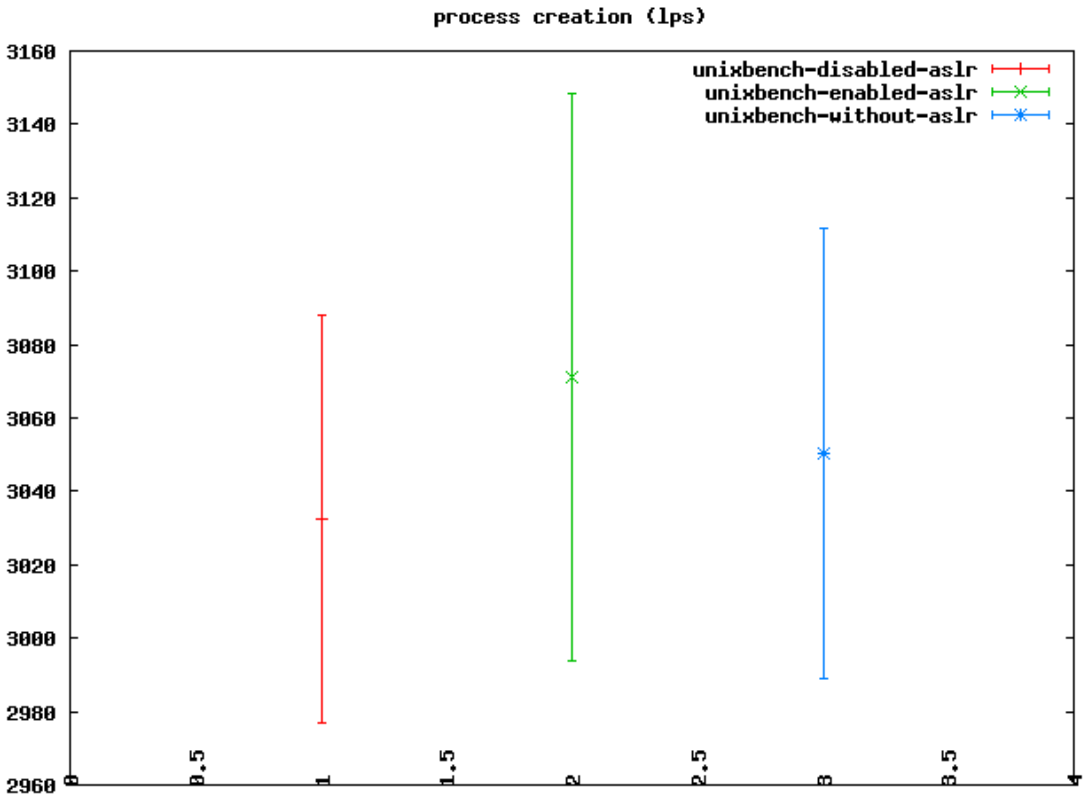


Figure 15: process creation (loops per sec)

A Figure 15: process creation (loops per sec) ábrán a vártakkal ellentétben a leggyorsabb a bekapcsolt ASLR-rel rendelkező kernel volt, azonban itt a mérési eredémény bizonytalan, mint ahogy az a szórásokból is látszik. Második lett a patch nélküli kernel és harmadik a patchelt, de kikapcsolt ASLR-rel futó kernel.

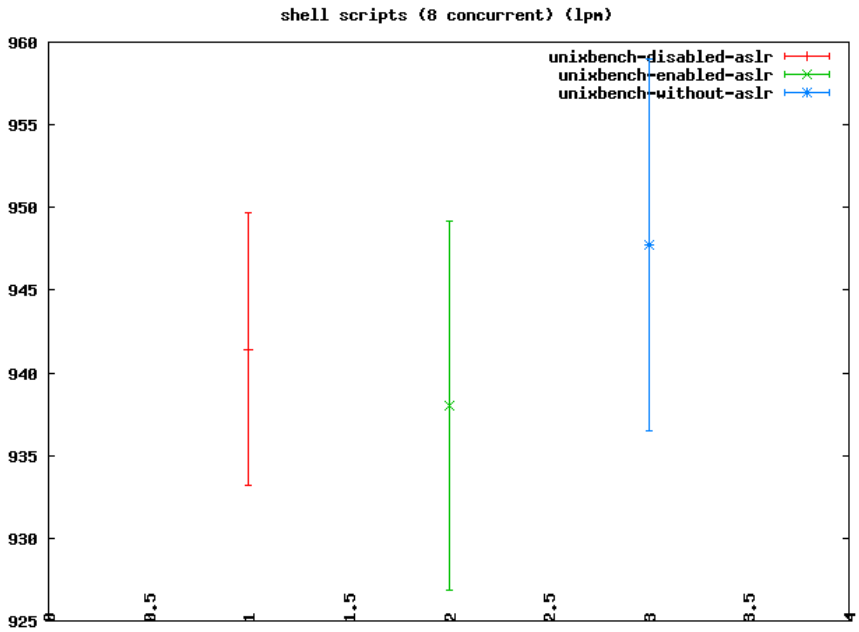


Figure 16: shell scripts (8 concurrent) (loops per sec)

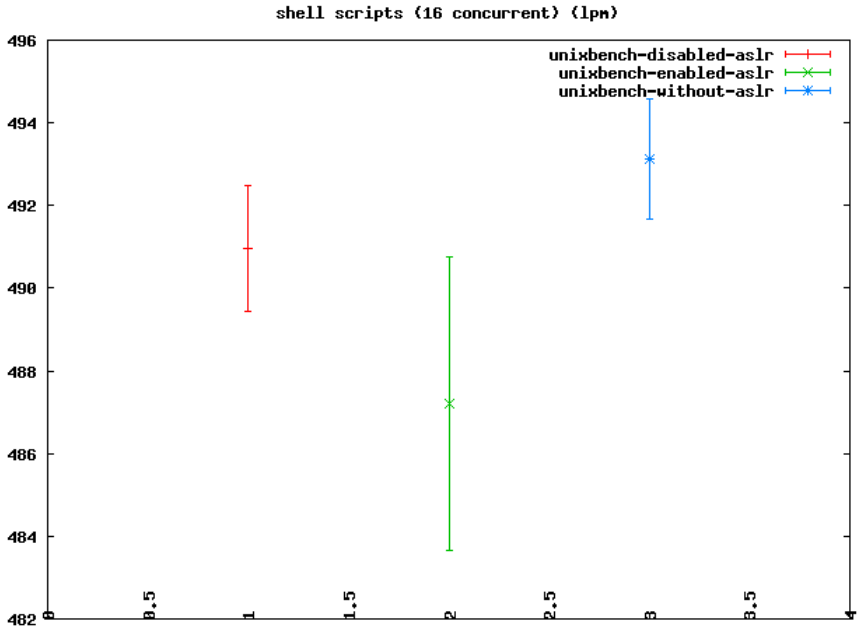


Figure 17: shell scripts (16 concurrent) (loops per sec)

A Figure 16: shell scripts (8 concurrent) (loops per sec) és Figure 17: shell scripts (16 concurrent) (loops per sec) ábrákon a Figure 14: execl throughput (loops per sec) ábránál látottakkal hasonló papírforma látható.

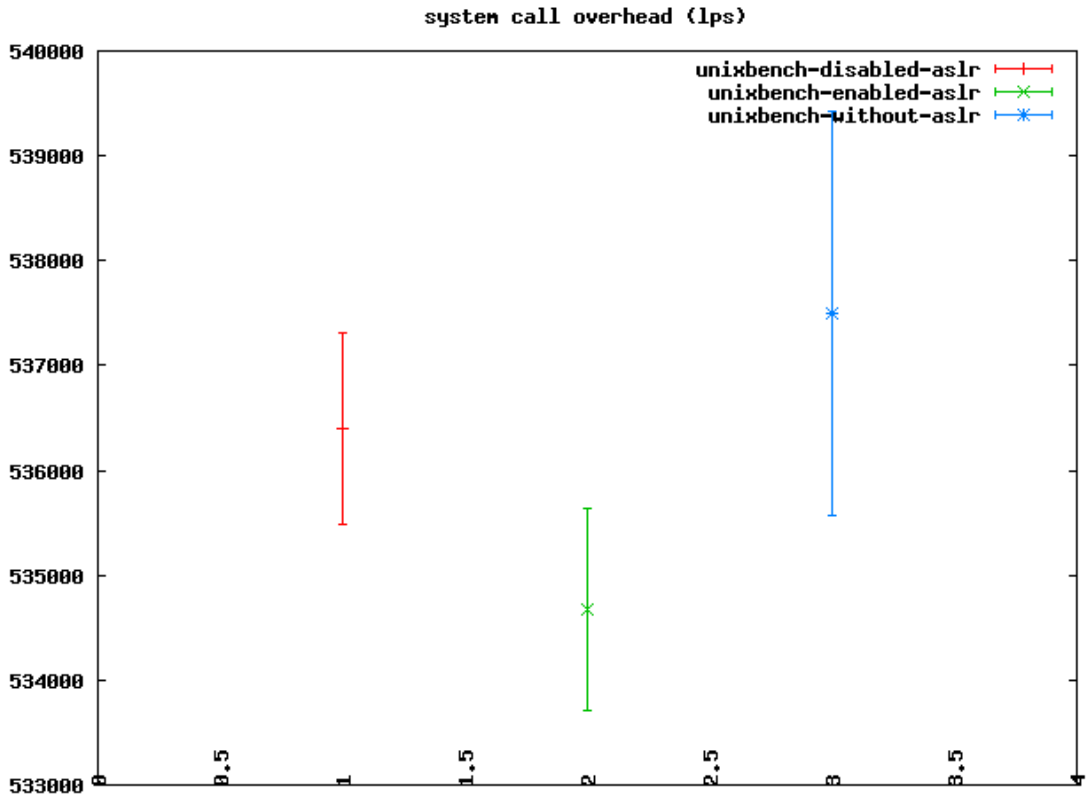


Figure 18: system call overhead (loops per sec)

A Figure 18: system call overhead (loops per sec) ábrán is a papírforma igazolódott be.

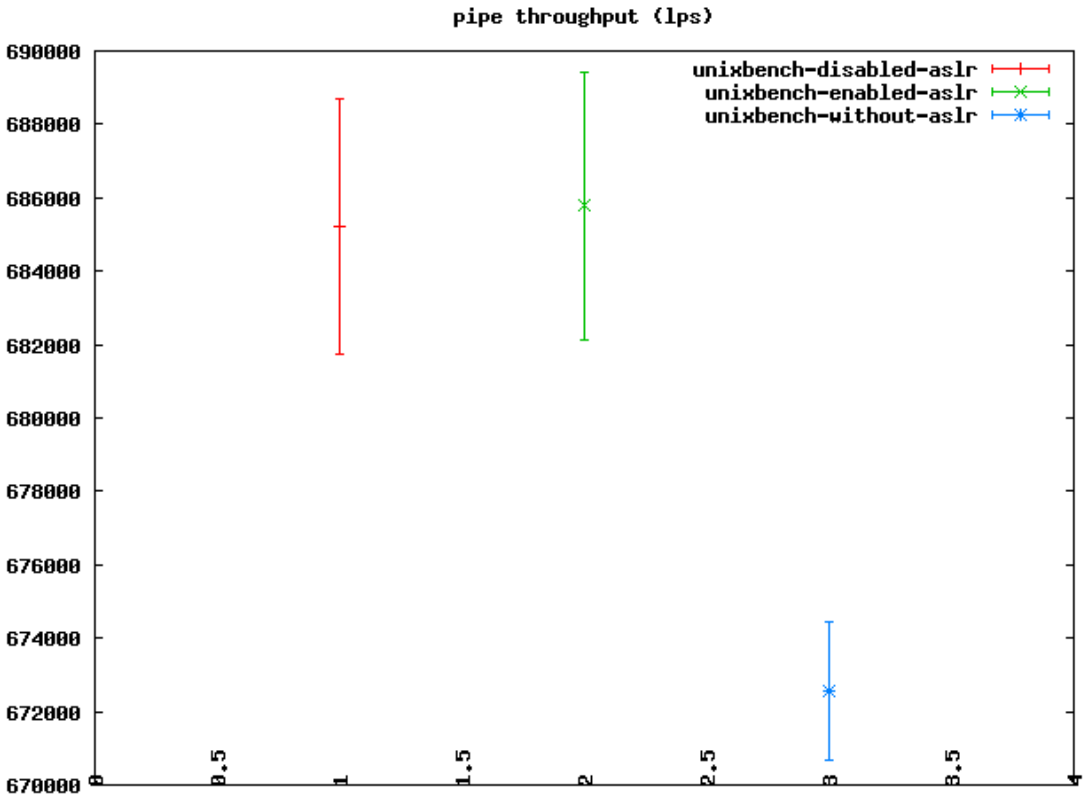


Figure 19: pipe throughput (loops per sec)

És a végére egy érdekesség, a Figure 19: pipe throughput (loops per sec) ábrán látható esetben mind kikapcsolt ASLR-rel, mind bekapcsolt ASLR-rel megnövekedett a rendszer *pipe(...)* megvalósításának áteresztőképessége.

Erre magyarázatot még nem találtam – megérzés alapján memoria alignment van a háttérben – további kutatást igényel a pontos ok kiderítése.

teszteset	*disabled vs without*	*enabled vs without*
C Compiler Throughput	99.76%	100.07%
Dc: sqrt(2) to 99 decimal places	100.30%	99.75%
Dhrystone 2 using register variables	103.31%	102.44%
Double-Precision Whetstone	99.97%	99.52%
Execl Throughput	99.36%	99.22%
File Copy 1024 bufsize 2000 maxblocks	99.26%	100.37%
File Copy 256 bufsize 500 maxblocks	100.75%	100.06%
File Copy 4096 bufsize 8000 maxblocks	99.98%	97.96%
File Read 1024 bufsize 2000 maxblocks	102.73%	102.70%
File Read 256 bufsize 500 maxblocks	101.75%	101.83%
File Read 4096 bufsize 8000 maxblocks	102.97%	102.39%
File Write 1024 bufsize 2000 maxblocks	97.14%	99.44%
File Write 256 bufsize 500 maxblocks	98.26%	98.23%
File Write 4096 bufsize 8000 maxblocks	100.94%	98.97%
Pipe Throughput	101.88%	101.96%
Pipe-based Context Switching	100.60%	102.80%
Process Creation	99.42%	100.67%
Recursion Test--Tower of Hanoi	99.73%	99.48%
Shell Scripts (1 concurrent)	99.48%	99.01%
Shell Scripts (16 concurrent)	99.56%	98.80%
Shell Scripts (8 concurrent)	99.33%	98.97%
System Call Overhead	99.80%	99.47%

A fenti táblázatban a kikapcsolt és a bekapcsolt ASLR viszonya látható a patchet nem tartalmzó kernelhez.

4.3 FreeBSD helyzete

Szándékosan hagytam a végére az állapotfelmérés összefoglalását, mivel nyomatékosítom, hogy a FreeBSD helyzete biztonsági téren messze el van maradva a többi rendszertől.

A 2.3 Memóriavédelmi megoldások fejezetben bemutatott egyes védelmek végén már adtam egy állapotjelentést, hogy az adott technológiát tartalmazza-e a FreeBSD. Ebben a fejezetben pedig összefoglalom, hogy legyen egy átlátható kép.

A státusz ebben az esetben az éppen aktuális fejlesztői változatra érvényes, mely a vizsgálatom alatt a FreeBSD 10-CURRENT r250423.

Technológia	*státusz*	*hardware feltétel*
PaX MPROTECT	nem tartalmazza	nem
W^X	nem tartalmazza	nem
XD	tartalmazza	igen
Stack canary (SSP)	tartalmazza	nem
ASLR	nem tartalmazza	nem
Segvguard	nem tartalmazza	nem
UDEREF	nem tartalmazza	nem
KERNEXEC	nem tartalmazza	nem
SMEP	tartalmazza	igen
SMAP	nem tartalmazza	igen

Ahogy a fenti táblázatból is látszik, a FreeBSD igen kevés memóriavédelmi megoldást implementál, és amelyek implementálva vannak, azok többsége hardware meglétéhez vannak kötve.

Ez annak a következménye, hogy a FreeBSD projekt a teljesítményt és a felhasználhatóságot tűzte a zászlajára, azonban e tény mellett az is közrejátszik, hogy bizonyos feladatok elvégzésére még így is kevés a fejlesztői kapacitás.

Bizonyos előremozdulás azonban látszik, mivel a fenti táblázatban található megoldások egy része az utóbbi évek termése.

Végezetül álljon itt pár *paxtest* kimenet, mely egy jövőképet vázolhat fel. A kiemeléseket célszerű figyelni.

paxtest - FreeBSD 9.0-RELEASE amd64 [45]

```
FreeBSD fbsd9amd64 9.0-RELEASE FreeBSD 9.0-RELEASE #0: Tue Jan  3 07:46:30
UTC  2012         root@farrell.cse.buffalo.edu:/usr/obj/usr/src/sys/GENERIC
amd64

Executable anonymous mapping             : Killed
Executable bss                           : Killed
Executable data                          : Killed
Executable heap                          : Killed
Executable stack                         : Vulnerable
Executable anonymous mapping (mprotect)  : Vulnerable
Executable bss (mprotect)                : Vulnerable
Executable data (mprotect)               : Vulnerable
Executable heap (mprotect)               : Vulnerable
Executable shared library bss (mprotect) : Vulnerable
Executable shared library data (mprotect): Vulnerable
Executable stack (mprotect)              : Vulnerable
Anonymous mapping randomisation test     : No randomisation
Heap randomisation test (ET_EXEC)        : No randomisation
Main executable randomisation (ET_EXEC)  : No randomisation
Shared library randomisation test        : No randomisation
Stack randomisation test (SEGMEXEC)      : No randomisation
Stack randomisation test (PAGEEXEC)      : No randomisation
Return to function (strcpy)                    : paxtest: return address
contains a NULL byte.
Return to function (strcpy, PIE)               : paxtest: return address
contains a NULL byte.
Return to function (memcpy)              : Vulnerable
Return to function (memcpy, PIE)         : Vulnerable
Executable shared library bss            : Killed
Executable shared library data           : Killed
Writable text segments                   : Vulnerable
```

paxtest-freebsd – FreeBSD 10-CURRENT(r249952)

```
FreeBSD pandora-test 10.0-CURRENT FreeBSD 10.0-CURRENT #69 r+05fa55f: Wed
May     1     10:16:10     CEST     2013                        op@pandora-
test:/usr/obj/usr/home/op/git/freebsd-base.git.http/sys/OP amd64

Executable anonymous mapping             : Killed
Executable bss                           : Killed
Executable data                          : Killed
Executable heap                          : Killed
Executable stack                         : Killed
Executable anonymous mapping (mprotect)  : Vulnerable
Executable bss (mprotect)                : Vulnerable
Executable data (mprotect)               : Vulnerable
Executable heap (mprotect)               : Vulnerable
Executable shared library bss (mprotect) : Vulnerable
Executable shared library data (mprotect): Vulnerable
Executable stack (mprotect)              : Vulnerable
Anonymous mapping randomisation test     : No randomisation
Heap randomisation test (ET_EXEC)        : No randomisation
Main executable randomisation (ET_EXEC)  : No randomisation
Shared library randomisation test        : No randomisation
Stack randomisation test (SEGMEXEC)      : No randomisation
Stack randomisation test (PAGEEXEC)      : No randomisation
Arg/env randomisation test (SEGMEXEC)    : No randomisation
Arg/env randomisation test (PAGEEXEC)    : No randomisation
Return to function (strcpy)                    : paxtest: return address
contains a NULL byte.
Return to function (strcpy, PIE)               : paxtest: return address
contains a NULL byte.
Return to function (memcpy)              : Vulnerable
Return to function (memcpy, PIE)         : Vulnerable
Executable shared library bss            : Killed
Executable shared library data           : Killed
Writable text segments                   : Vulnerable
```

paxtest-freebsd – FreeBSD 10-CURRENT(r249952) + ASLR patch

```
FreeBSD pandora-test 10.0-CURRENT FreeBSD 10.0-CURRENT #80 r+70a9d86: Mon
May     13     16:48:55     CEST     2013                       op@pandora-
test:/usr/obj/usr/home/op/git/freebsd-base.git.http/sys/OP  amd64

Executable anonymous mapping             : Killed
Executable bss                           : Killed
Executable data                          : Killed
Executable heap                          : Killed
Executable stack                         : Killed
Executable anonymous mapping (mprotect)  : Vulnerable
Executable bss (mprotect)                : Vulnerable
Executable data (mprotect)               : Vulnerable
Executable heap (mprotect)               : Vulnerable
Executable shared library bss (mprotect) : Vulnerable
Executable shared library data (mprotect): Vulnerable
Executable stack (mprotect)              : Vulnerable
Anonymous mapping randomisation test     : 16 bits (guessed)
Heap randomisation test (ET_EXEC)        : 7 bits (guessed)
Main executable randomisation (ET_EXEC)  : No randomisation
Shared library randomisation test        : 17 bits (guessed)
Stack randomisation test (SEGMEXEC)      : 12 bits (guessed)
Stack randomisation test (PAGEEXEC)      : 12 bits (guessed)
Arg/env randomisation test (SEGMEXEC)    : 13 bits (guessed)
Arg/env randomisation test (PAGEEXEC)    : 13 bits (guessed)
Return to function (strcpy)                    : paxtest: return address
contains a NULL byte.
Return to function (strcpy, PIE)               : paxtest: return address
contains a NULL byte.
Return to function (memcpy)              : Vulnerable
Return to function (memcpy, PIE)         : Vulnerable
Executable shared library bss            : Killed
Executable shared library data           : Killed
Writable text segments                   : Vulnerable
```

Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani technológiai review-kért és tanácsokért: Hunger-nek <hunger@hunger.hu>, konzulensemnek Bencsáth Boldizsárnak <boldizsar.bencsath@crysys.hit.bme.hu>, Konstantin Belousov <kib@freebsd.org> FreeBSD fejlesztőnek, PaX Team-nek <pageexec@freemail.hu>.

Tesztelésért: Hiren Panchasara-nak <hiren.panchasara@gmail.com> Yahoo FreeBSD fejlesztőnek, Fodor Zoltánnak <zoltan.fodor@intel.com>, Pawel Gepner <pawel.gepner@intel.com>.

Általános review-kért és javításokért: Bordás Katalinnak, Lágler Gergelynek, László Blankának és Morvai Boglárkának.

Végezetül, de nem utolsó sorban: édesapámnak Pintér Sándornak, családomnak és barátaimnak a támogatásért.

Irodalomjegyzék

[1] Intel: Intel® 64 and IA-32 Architectures Software Developer Manuals - Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C, Order Number: 325462-046US, March 2013, http://download.intel.com/products/processor/manual/325462.pdf

[2] Intel: Intel® Architecture Instruction Set Extensions Programming Reference, 319433-014, AUGUST 2012, Chapter 9.3, http://software.intel.com/sites/default/files/319433-014.pdf

[3] Intel: Intel® Architecture Instruction Set Extensions Programming Reference, 319433-014, AUGUST 2012, Chapter 9.6, table 9-13, table 9-14 http://software.intel.com/sites/default/files/319433-014.pdf

[4] Intel: Intel® 64 and IA-32 Architectures Software Developer Manuals - Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C, Order Number: 325462-046US, March 2013, SDM vol.3 8.1.3, http://download.intel.com/products/processor/manual/325462.pdf

[5] Intel: Intel® 64 and IA-32 Architectures Software Developer Manuals - Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C, Order Number: 325462-046US, March 2013, SDM vol.3 11.6, http://download.intel.com/products/processor/manual/325462.pdf

[6] Marshall Kirk McKusick, George V. Neville-Neil: The Design and Implementation of the FreeBSD Operating System, 2.2 Kernel Organization, http://books.google.hu/books/about/The_Design_and_Implementation_of_the_Fre.html?id=-RqMzqH__BMC&redir_esc=y

[7] FreeBSD: FreeBSD Man Pages – sysctl(8). http://www.freebsd.org/cgi/man.cgi?query=sysctl&sektion=8

[8] FreeBSD: FreeBSD Man Pages – sysctl(3). http://www.freebsd.org/cgi/man.cgi?query=sysctl&sektion=3&apropos=0&manpath=FreeBSD+9.1-RELEASE

[9] Intel: Intel® 64 and IA-32 Architectures Software Developer Manuals - Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C, Order Number: 325462-046US, March 2013, SDM vol.3 4.1.4, http://download.intel.com/products/processor/manual/325462.pdf

[10] PaX Team: Address Space Layout Randomization, http://pax.grsecurity.net/docs/aslr.txt

[11] PaX Team: RANDMMAP, http://pax.grsecurity.net/docs/randmmap.txt

[12] PaX Team: RANDUSTACK, http://pax.grsecurity.net/docs/randustack.txt

[13] PaX Team: RANDKSTACK, http://pax.grsecurity.net/docs/randkstack.txt

[14] PaX Team: UDEREF, http://grsecurity.net/~spender/uderef.txt

[15] PaX Team: UDEREF64, http://grsecurity.net/pipermail/grsecurity/2010-April/001024.html

[16] Nergal: The advanced return-into-lib(c) exploits: PaX case study, http://www.phrack.org/issues.html?issue=58&id=4&mode=txt

[17] Tyler Durden: Bypassing PaX ASLR protection, http://www.theparticle.com/files/txt/hacking/phrack/p59-0x09.txt

[18] LWN: SMP alternatives, http://lwn.net/Articles/164121/

[19] PaX Team: SMAP, http://forums.grsecurity.net/viewtopic.php?f=7&t=3046

[20] FreeBSD: FreeBSD Kernel Interface Manual, ddb, http://www.freebsd.org/cgi/man.cgi?query=ddb&sektion=4&apropos=0&manpath=FreeBSD+9.1-RELEASE

[21] John H. Baldwin, Introduction to Debugging the FreeBSD Kernel, http://www.bsdcan.org/2008/schedule/attachments/45_article.pdf

[22] FreeBSD: FreeBSD General Commands Manual, kgdb. http://www.freebsd.org/cgi/man.cgi?query=kgdb&apropos=0&sektion=0&manpath=FreeBSD+9.1-RELEASE&arch=default&format=html

[23] Pintér Olivér: SMAP tester, https://github.com/opntr/freebsd-smap-tester

[24] Marshall Kirk McKusick, George V. Neville-Neil: The Design and Implementation of the FreeBSD Operating System, 5.2. Overview of the FreeBSD Virtual-Memory System, http://www.ico.aha.ru/h/The_Design_and_Implementation_of_the_FreeBSD_Operating_System/ch05lev1sec2.htm

[25] FreeBSD System Calls Manual, mmap(2), http://www.freebsd.org/cgi/man.cgi?query=mmap&apropos=0&sektion=0&manpath=FreeBSD+9.1-RELEASE&arch=default&format=html

[26] Jan Hubicka, Andreas Jaeger, Mark Mitchell: System V Application Binary Interface, AMD64 Architecture Processor Supplement, http://people.freebsd.org/~obrien/amd64-elf-abi.pdf

[27] Felix FX Lindner, A heap of risk, http://www.h-online.com/security/features/A-Heap-of-Risk-747161.html

[28] Ryan Roemer, Erik Buchanan, Hovav Shacham and Stefan Savage: Return-Oriented Programming: Systems, Languages, and Applications, http://cseweb.ucsd.edu/~hovav/dist/rop.pdf

[29] Sebastian Krahmer: x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique, http://users.suse.com/~krahmer/no-nx.pdf

[30] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, Zhenkai Liang: Jump-Oriented Programming: A New Class of Code-Reuse Attack, http://www.comp.nus.edu/~liangzk/papers/asiaccs11.pdf

[31] Ralf Hund, Carsten Willems, Thorsten Holz: Practical Timing Side Channel Attacks Against Kernel Space ASLR, http://www.internetsociety.org/sites/default/files/Practical%20Timing%20Side%20Channel%20Attacks%20Against%20Kernel%20Space%20ASLR.pdf

[32] OWASP: Format string attack, https://www.owasp.org/index.php/Format_string_attack

[33] Gera, Riq: Advances in format string exploitation, http://www.phrack.org/issues.html?issue=59&id=7&mode=txt

[34] FreeBSD: FreeBSD Man Pages – sysctl(9), http://www.freebsd.org/cgi/man.cgi?query=sysctl&sektion=9&manpath=FreeBSD+9.1-RELEASE

[35] Spender, PaX Team: KASLR – KASLR: An Exercise in Cargo Cult Security, http://forums.grsecurity.net/viewtopic.php?f=7&t=3367

[36] FreeBSD: FreeBSD Man Pages – procstat(1), http://www.freebsd.org/cgi/man.cgi?procstat

[37] Spender:paxtest, http://grsecurity.net/~spender/paxtest-0.9.11.tar.gz

[38] Oliver Pinter: paxtest-freebsd, https://github.com/opntr/paxtest-freebsd

[39] Byte Magazine: unixbench, http://code.google.com/p/byte-unixbench/

[40] PaX Team: MPROTECT, http://pax.grsecurity.net/docs/mprotect.txt

[41] Hiroaki Etoh: ProPolice, http://pacsec.jp/psj04/psj04-hiroaki-e.ppt

[42] The de Raadt: Exploit Mitigation Techniques, http://www.openbsd.org/papers/ven05-deraadt/index.html

[43] Spender: PaX -The Guaranteed End of Arbitrary Code Execution, http://grsecurity.net/PaX-presentation.ppt

[44] Intel: Intel® 64 and IA-32 Architectures Software Developer Manuals - Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C, Order Number: 325462-046US, March 2013, page 1767, http://download.intel.com/products/processor/manual/325462.pdf

[45] Hunger: paxtest-freebsd9amd64.txt, http://hunger.hu/paxtest-freebsd9amd64.txt

[46] Netcraft: Most Reliable Hosting Company Site in April 2013, http://news.netcraft.com/archives/2013/05/01/most-reliable-hosting-company-sites-in-april-2013-2.html

[47] FreeBSD: FreeBSD problem reports, http://www.freebsd.org/cgi/query-pr.cgi?pr=ports/177488

[48] WikiBooks: Grsecurity/Appendix/Grsecurity and PaX Configuration Options, http://en.wikibooks.org/wiki/Grsecurity/Appendix/Grsecurity_and_PaX_Configuration_Options

[49] PaX Team: PaX - kernel self-protection, http://pax.grsecurity.net/docs/PaXTeam-H2HC12-PaX-kernel-self-protection.pdf

[50] PaX Team: pageexec, http://pax.grsecurity.net/docs/pageexec.txt

Ábrajegyzék

Rövidítések

ASLR	Address Space Layout Randomization
BIOS	Basic Input/Output System
BSD	Berkeley Software Distribution
BSM	Basic Security Module
CLAC	CLear AC flag
CPL	Current Privilege Level
CPU	Central Processing Unit
DAC	Discretionary Access Control
DARPA	Defense Advenced Research Projects Agency
ELF	Executable and Linkable Format
ISP	Internet Service Provider
IT	Information Technology
JIT	Just-In-Time
JOP	Jump-Oriented Programming
KASSERT	Kernel Assert
lpm	loops per minute
lps	loops per sec
MAC	Mandatory Access Control
MD	Machine-Dependent
MI	Machine-Independent
MWIPS	Millions of Whetstone Instructions Per Second
NBBY	Number of Bits per Byte
NOP	No Operation

NX	Never eXecute,
PIC	Position-Independent Code
PID	Process ID
PIE	Position-Independent Executable
ROP	Return-Oriented Programming
RTLD	Run-Time Link eDitor
SDM	Software Developers's Manual
SMAP	Supervisor-Mode Access Prevention
SMEP	Supervisor-Mode Execution Prevention
SSP	Stack Smash Protection
STAC	SeT AC flag
TCG	Tiny Code Generator
TID	Thread ID
UFS	Unix File System
VM	Virtual Memory
XD	eXecute Disable

Függelék

Az SMAP technológia teszteléséhez használt emulátor neve Qemu – ez egy CPU emulátor. A fejlesztések kezdetekor az 1.4-es verzió még nem volt bent a FreeBSD ports rendszerben, így azt nekem kellett működésre bírnom. Ebből született pár FreeBSD-s bug report[47] és patch. Miután működésre bírtam az emulátort, elkezdődhetett a tesztelés – időközben a FreeBSD ports-ban is frissült az emulátor 1.4-es verzióra.

Ezen felül az emulátort TCG (tiny code generator) támogatással kell fordítani, mivel ez emulálja a host CPU által nem támogatott utasításokat és szolgáltatásokat, mint például az SMAP-ot.

Az emulátor rendkívül konfigurálható és különböző processzorcsaládokat támogat:

```
op@pandora-test paxtest-freebsd# qemu-system-x86_64 -cpu \?
x86           qemu64  QEMU Virtual CPU version 1.4.1
x86           phenom  AMD Phenom(tm) 9550 Quad-Core Processor
x86         core2duo  Intel(R) Core(TM)2 Duo CPU     T7700  @ 2.40GHz
x86            kvm64  Common KVM processor
x86           qemu32  QEMU Virtual CPU version 1.4.1
x86            kvm32  Common 32-bit KVM processor
x86          coreduo  Genuine Intel(R) CPU           T2600  @ 2.16GHz
x86              486
x86          pentium
x86         pentium2
x86         pentium3
x86           athlon  QEMU Virtual CPU version 1.4.1
x86             n270  Intel(R) Atom(TM) CPU N270   @ 1.60GHz
x86           Conroe  Intel Celeron_4x0 (Conroe/Merom Class Core 2)
x86           Penryn  Intel Core 2 Duo P9xxx (Penryn Class Core 2)
x86          Nehalem  Intel Core i7 9xx (Nehalem Class Core i7)
x86         Westmere  Westmere E56xx/L56xx/X56xx (Nehalem-C)
x86      SandyBridge  Intel Xeon E312xx (Sandy Bridge)
x86          Haswell  Intel Core Processor (Haswell)
x86       Opteron_G1  AMD Opteron 240 (Gen 1 Class Opteron)
x86       Opteron_G2  AMD Opteron 22xx (Gen 2 Class Opteron)
x86       Opteron_G3  AMD Opteron 23xx (Gen 3 Class Opteron)
x86       Opteron_G4  AMD Opteron 62xx class CPU
x86       Opteron_G5  AMD Opteron 63xx class CPU
```

Esetemben a Haswell CPU architektúra kiválasztása lett volna célszerű, de ebben az esetben az SMAP szolgáltatás nem volt elérhető legnagyobb meglepetésemre. Az interneten történő hosszabb kutatás sem járt nagyobb eredménnyel, mint hogy a Qemu 1.4-től kezdve támogatva van az SMAP és egy patch-re bukkantam különösebb

leírások nélkül. Ezt követően a Qemu forrását választottam dokumentációként és megkerestem, hogy milyen esetekben működhet biztosan az SMAP. Ezt követően létrehoztam egy teszt script-et, mivel a fejlesztés során számtalanszor kellett használni ezt a programot általában azonos beállításokkal.

```
op@pandora-test git# less smap_test.csh
#!/bin/csh

set FREEBSD_PREFIX="/usr/home/op/git/"
#set QEMU_PREFIX="/usr/home/op/qemu/"
set QEMU_PREFIX="/usr/local/"

set QEMU="${QEMU_PREFIX}/bin/qemu"
set QEMU_X86_64="${QEMU_PREFIX}/bin/qemu-system-x86_64"
set QEMU_OPTIONS="-cpu qemu64,enforce,+smap,-hypervisor -m 1024M"

set HDD_IMAGE="${FREEBSD_PREFIX}/freebsd-test.raw"

$QEMU_X86_64 $QEMU_OPTIONS -hda $HDD_IMAGE
```

Ahogy a vastagon kiemelt sorban látható *Haswell* CPU típus helyett *qemu64*-et használok, mely a Qemu saját metaprocesszora, amit széles körben lehet konfigurálni.

Három lényeges opciót kellett megadni a CPU típusán kívül:

- enforce
- +smap
- -hypervisor

Az enforce nevéből adódik, hogy az adott opciókat mindenképp be akarjuk kapcsolni.

A +smap opcióval jelezzük, hogy az SMAP szolgáltatást be akarjuk kapcsolni.

A –hypervisor pedig egy FreeBSD specifikus beállítás. Kikapcsolja a hypervisor bitet, mivel a Qemu alapértelmezetten beállítja az emulált processzorban. Azonban a FreeBSD és az általam készített SMAP implementáció is kikapcsol bizonyos szolgáltatásokat hypervisor-ok alatt, melyet ezen bit segítségével érzékel. Erre azért van szükség, mivel néhány hypervisor nem szűri ki megfelelően a nem támogatott kiterjesztéseket a FreeBSD-ben található komment szerint.

Az smap_test.csh script egy virtuális gépet indít el, melyben egy FreeBSD 10-CURRENT kapott helyet.

Az image frissítése ugyancsak egy script segítségével történik, melyet az smap_test.csh után mutatok be.

update_freebsd.csh:

```
#!/bin/csh

set FREEBSD_PREFIX="/usr/home/op/git/"
set QEMU_PREFIX="/usr/home/op/qemu/"
set MDCONFIG="/sbin/mdconfig"
set QEMU="${QEMU_PREFIX}/bin/qemu"
set QEMU_IMG="${QEMU_PREFIX}bin/qemu-img"
set QEMU_X86_64="${QEMU_PREFIX}/bin/qemu-system-x86_64"
set QEMU_OPTIONS="-cpu Haswell -m 1024M"
set FSCK="/sbin/fsck_ffs"
set HDD_IMAGE="${FREEBSD_PREFIX}/freebsd-test.raw"

if (! -e $HDD_IMAGE) then
        ${FREEBSD_PREFIX}/mkenv_haswell_env.csh
endif

set MD_DEV=`$MDCONFIG -a -t vnode -f $HDD_IMAGE`
if (! -e /dev/$MD_DEV) then
        echo "mdconfig failed..."
        exit -1
endif

set MD_DEV_ROOT="/dev/${MD_DEV}p2"
$FSCK -y $MD_DEV_ROOT

setenv DESTDIR ${FREEBSD_PREFIX}/target
if (! -e $DESTDIR) then
        mkdir $DESTDIR
endif

mount /dev/${MD_DEV}p2 $DESTDIR

cd ${FREEBSD_PREFIX}/freebsd-base.git.http
#make -j4 buildworld installworld kernel  -DNO_CLEAN
make -j4 kernel -DNO_CLEAN || make -j4 kernel

umount ${FREEBSD_PREFIX}/target

$MDCONFIG -d -u $MD_DEV
```

A script röviden leírva, egy FreeBSD image file-t csatol fel egy speciális alrendszeren keresztül, melynek a neve *md*. Ezt követően az image file helyett egy *block eszközt* fogunk látni a */dev* könyvtárban, melynek nevével az *mdconfig* eszköz visszatér. Utána az ezen az eszközön található *root partíciót* ellenőrizzük, mivel a tesztelés alatt kernel *panic(...)*-ot kapunk számtalanszor és ekkor a file rendszer inkonzisztens állapotba kerül. Ha ez lefutott, akkor felcsatoljuk a root file rendszert a *target* mappába és lebuildeljük az új kernel-t a *make kernel* parancs segítségével. Ha ez lefutott és

frissítette a kernel-t, akkor a file rendszert lecsatoljuk, és az mdconfig segítségével lecsatoljuk az image-t is az md alrendszerrel. Ezt követően az először bemutatott script segítségével mehet a tesztelés.

A tesztelés nem lett automatizálva, mivel kézi beavatkozást és elemzést is számtalanszor igényel.

Az image-re elhelyeztem a már korábban említett smap-tester programokat és kézzel futtattam őket.

A kezdeti szakaszban, amikor a ksp alrendszer fejlesztésével foglalkoztam, gyorsabbnak láttam két fizikai FreeBSD gép használatát. Az egyik gép (laptop) volt a debugger gép a másik (desktop) pedig a teszt gép.

A két gép között soros kapcsolat volt.

Mivel az identcpu és initcpu szakaszban a kernel még kevésbé működőképes és USB perifériák nem működnek. A kernel viszont nyújt soros terminál szolgáltatást, amin keresztül a kernelt DDB-vel egyszerűen lehetett debuggolni.

Ehhez az alábbi sorok beírására volt szükség a /boot/loader.conf file-ba:

```
boot_multicons="YES"
boot_serial="YES"
console="vidconsole,comconsole"
```