```
**********************************************************
   16334 Tue Jan 27 15:02:14 2015
new/usr/src/cmd/mdb/intel/mdb/mdb_amd64util.c
5554 kmdb can't trace stacks that begin within itself
Reviewed by: Josef 'Jeff' Sipek <jeffpc@josefsipek.net>
**********************************************************
_____unchanged_portion_omitted_

 198 int
 199 mdb_amd64_kvm_stack_iter(mdb_tgt_t *t, const mdb_tgt_gregset_t *gsp,
 200     mdb_tgt_stack_f *func, void *arg)
 201 {
 202         mdb_tgt_gregset_t gregs;
 203         kreg_t *kregs = &gregs.kregs[0];
 204         int got_pc = (gsp->kregs[KREG_RIP] != 0);
 205         uint_t argc, reg_argc;
 206         long fr_argv[32];
 207         int start_index; /* index to save_instr where to start comparison */
 208         int err;
 209         int i;

 211         struct fr {
 211         struct {
 212                 uintptr_t fr_savfp;
 213                 uintptr_t fr_savpc;
 214         } fr;

 216         uintptr_t fp = gsp->kregs[KREG_RBP];
 217         uintptr_t pc = gsp->kregs[KREG_RIP];
 218         uintptr_t lastfp = 0;

 220         ssize_t size;
 221         ssize_t insnsize;
 222         uint8_t ins[SAVEARGS_INSN_SEQ_LEN];

 224         GElf_Sym s;
 225         mdb_syminfo_t sip;
 226         mdb_ctf_funcinfo_t mfp;
 227         int xpv_panic = 0;
 228         int advance_tortoise = 1;
 229         uintptr_t tortoise_fp = 0;
 230 #endif /* ! codereview */
 231 #ifndef _KMDB
 232         int xp;

 234         if ((mdb_readsym(&xp, sizeof (xp), "xpv_panicking") != -1) && (xp > 0))
 235                 xpv_panic = 1;
 236 #endif

 238         bcopy(gsp, &gregs, sizeof (gregs));

 240         while (fp != 0) {
 241                 int args_style = 0;

 243                 if (mdb_tgt_vread(t, &fr, sizeof (fr), fp) != sizeof (fr)) {
 244                         err = EMDB_NOMAP;
 228                 /*
 229                  * Ensure progress (increasing fp), and prevent
 230                  * endless loop with the same FP.
 231                  */
 232                 if (fp <= lastfp) {
 233                         err = EMDB_STKFRAME;
 245                         goto badfp;
 246                 }

 248                 if (tortoise_fp == 0) {
```



```
 249                         tortoise_fp = fp;
 250                 } else {
 251                         if (advance_tortoise != 0) {
 252                                 struct fr tfr;

 254                                 if (mdb_tgt_vread(t, &tfr, sizeof (tfr),
 255                                     tortoise_fp) != sizeof (tfr)) {
 236                 if (mdb_tgt_vread(t, &fr, sizeof (fr), fp) != sizeof (fr)) {
 256                                         err = EMDB_NOMAP;
 257                                         goto badfp;
 258                                 }

 260                                 tortoise_fp = tfr.fr_savfp;
 261                         }

 263                         if (fp == tortoise_fp) {
 264                                 err = EMDB_STKFRAME;
 265                                 goto badfp;
 266                         }
 267                 }

 269                 advance_tortoise = !advance_tortoise;

 271 #endif /* ! codereview */
 272                 if ((mdb_tgt_lookup_by_addr(t, pc, MDB_TGT_SYM_FUZZY,
 273                     NULL, 0, &s, &sip) == 0) &&
 274                     (mdb_ctf_func_info(&s, &sip, &mfp) == 0)) {
 275                         int return_type = mdb_ctf_type_kind(mfp.mtf_return);
 276                         mdb_ctf_id_t args_types[5];

 278                         argc = mfp.mtf_argc;

 280                         /*
 281                          * If the function returns a structure or union
 282                          * greater than 16 bytes in size %rdi contains the
 283                          * address in which to store the return value rather
 284                          * than for an argument.
 285                          */
 286                         if ((return_type == CTF_K_STRUCT ||
 287                             return_type == CTF_K_UNION) &&
 288                             mdb_ctf_type_size(mfp.mtf_return) > 16)
 289                                 start_index = 1;
 290                         else
 291                                 start_index = 0;

 293                         /*
 294                          * If any of the first 5 arguments are a structure
 295                          * less than 16 bytes in size, it will be passed
 296                          * spread across two argument registers, and we will
 297                          * not cope.
 298                          */
 299                         if (mdb_ctf_func_args(&mfp, 5, args_types) == CTF_ERR)
 300                                 argc = 0;

 302                         for (i = 0; i < MIN(5, argc); i++) {
 303                                 int t = mdb_ctf_type_kind(args_types[i]);

 305                                 if (((t == CTF_K_STRUCT) ||
 306                                     (t == CTF_K_UNION)) &&
 307                                     mdb_ctf_type_size(args_types[i]) <= 16) {
 308                                         argc = 0;
 309                                         break;
 310                                 }
 311                         }
 312                 } else {
 313                         argc = 0;
```

```
314                 }
315
316                 /*
317                  * The number of instructions to search for argument saving is
318                  * limited such that only instructions prior to %pc are
319                  * considered such that we never read arguments from a
320                  * function where the saving code has not in fact yet
321                  * executed.
322                  */
323                 insnsize = MIN(MIN(s.st_size, SAVEARGS_INSN_SEQ_LEN),
324                     pc - s.st_value);
325
326                 if (mdb_tgt_vread(t, ins, insnsize, s.st_value) != insnsize)
327                         argc = 0;
328
329                 if ((argc != 0) &&
330                     ((args_style = saveargs_has_args(ins, insnsize, argc,
331                     start_index)) != SAVEARGS_NO_ARGS)) {
332                         /* Up to 6 arguments are passed via registers */
333                         reg_argc = MIN((6 - start_index), mfp.mtf_argc);
334                         size = reg_argc * sizeof (long);
335
336                         /*
337                          * If Studio pushed a structure return address as an
338                          * argument, we need to read one more argument than
339                          * actually exists (the addr) to make everything line
340                          * up.
341                          */
342                         if (args_style == SAVEARGS_STRUCT_ARGS)
343                                 size += sizeof (long);
344
345                         if (mdb_tgt_vread(t, fr_argv, size, (fp - size))
346                             != size)
347                                 return (-1);    /* errno has been set for us */
348
349                         /*
350                          * Arrange the arguments in the right order for
351                          * printing.
352                          */
353                         for (i = 0; i < (reg_argc / 2); i++) {
354                                 long t = fr_argv[i];
355
356                                 fr_argv[i] = fr_argv[reg_argc - i - 1];
357                                 fr_argv[reg_argc - i - 1] = t;
358                         }
359
360                         if (argc > reg_argc) {
361                                 size = MIN((argc - reg_argc) * sizeof (long),
362                                     sizeof (fr_argv) -
363                                     (reg_argc * sizeof (long)));
364
365                                 if (mdb_tgt_vread(t, &fr_argv[reg_argc], size,
366                                     fp + sizeof (fr)) != size)
367                                         return (-1); /* errno has been set */
368                         }
369                 } else {
370                         argc = 0;
371                 }
372
373                 if (got_pc && func(arg, pc, argc, fr_argv, &gregs) != 0)
374                         break;
375
376                 kregs[KREG_RSP] = kregs[KREG_RBP];
377
378                 lastfp = fp;
379                 fp = fr.fr_savfp;
```



```
380                 /*
381                  * The Xen hypervisor marks a stack frame as belonging to
382                  * an exception by inverting the bits of the pointer to
383                  * that frame.  We attempt to identify these frames by
384                  * inverting the pointer and seeing if it is within 0xfff
385                  * bytes of the last frame.
386                  */
387                 if (xpv_panic)
388                         if ((fp != 0) && (fp < lastfp) &&
389                             ((lastfp ^ ~fp) < 0xfff))
390                                 fp = ~fp;
391
392                 kregs[KREG_RBP] = fp;
393                 kregs[KREG_RIP] = pc = fr.fr_savpc;
394
395                 got_pc = (pc != 0);
396         }
397
398         return (0);
399
400 badfp:
401         mdb_printf("%p [%s]", fp, mdb_strerror(err));
402         return (set_errno(err));
403 }
404
405 /*
406  * Determine the return address for the current frame.  Typically this is the
407  * fr_savpc value from the current frame, but we also perform some special
408  * handling to see if we are stopped on one of the first two instructions of
409  * a typical function prologue, in which case %rbp will not be set up yet.
410  */
411 int
412 mdb_amd64_step_out(mdb_tgt_t *t, uintptr_t *p, kreg_t pc, kreg_t fp, kreg_t sp,
413     mdb_instr_t curinstr)
414 {
415         struct frame fr;
416         GElf_Sym s;
417         char buf[1];
418
419         enum {
420                 M_PUSHQ_RBP     = 0x55, /* pushq %rbp */
421                 M_REX_W         = 0x48, /* REX prefix with only W set */
422                 M_MOVL_RBP      = 0x8b  /* movq %rsp, %rbp with prefix */
423         };
424
425         if (mdb_tgt_lookup_by_addr(t, pc, MDB_TGT_SYM_FUZZY,
426             buf, 0, &s, NULL) == 0) {
427                 if (pc == s.st_value && curinstr == M_PUSHQ_RBP)
428                         fp = sp - 8;
429                 else if (pc == s.st_value + 1 && curinstr == M_REX_W) {
430                         if (mdb_tgt_vread(t, &curinstr, sizeof (curinstr),
431                             pc + 1) == sizeof (curinstr) && curinstr ==
432                             M_MOVL_RBP)
433                                 fp = sp;
434                 }
435         }
436
437         if (mdb_tgt_vread(t, &fr, sizeof (fr), fp) == sizeof (fr)) {
438                 *p = fr.fr_savpc;
439                 return (0);
440         }
441
442         return (-1); /* errno is set for us */
443 }
444
445 /*ARGSUSED*/
```

```
446 int
447 mdb_amd64_next(mdb_tgt_t *t, uintptr_t *p, kreg_t pc, mdb_instr_t curinstr)
448 {
449         mdb_tgt_addr_t npc;
450         mdb_tgt_addr_t callpc;

452         enum {
453                 M_CALL_REL = 0xe8, /* call near with relative displacement */
454                 M_CALL_REG = 0xff, /* call near indirect or call far register */

456                 M_REX_LO = 0x40,
457                 M_REX_HI = 0x4f
458         };

460         /*
461          * If the opcode is a near call with relative displacement, assume the
462          * displacement is a rel32 from the next instruction.
463          */
464         if (curinstr == M_CALL_REL) {
465                 *p = pc + sizeof (mdb_instr_t) + sizeof (uint32_t);
466                 return (0);
467         }

469         /* Skip the rex prefix, if any */
470         callpc = pc;
471         while (curinstr >= M_REX_LO && curinstr <= M_REX_HI) {
472                 if (mdb_tgt_vread(t, &curinstr, sizeof (curinstr), ++callpc) !=
473                     sizeof (curinstr))
474                         return (-1); /* errno is set for us */
475         }

477         if (curinstr != M_CALL_REG) {
478                 /* It's not a call */
479                 return (set_errno(EAGAIN));
480         }

482         if ((npc = mdb_dis_nextins(mdb.m_disasm, t, MDB_TGT_AS_VIRT, pc)) == pc)
483                 return (-1); /* errno is set for us */

485         *p = npc;
486         return (0);
487 }

489 /*ARGSUSED*/
490 int
491 mdb_amd64_kvm_frame(void *arglim, uintptr_t pc, uint_t argc, const long *argv,
492     const mdb_tgt_gregset_t *gregs)
493 {
494         argc = MIN(argc, (uintptr_t)arglim);
495         mdb_printf("%a(", pc);

497         if (argc != 0) {
498                 mdb_printf("%lr", *argv++);
499                 for (argc--; argc != 0; argc--)
500                         mdb_printf(", %lr", *argv++);
501         }

503         mdb_printf(")\n");
504         return (0);
505 }

507 int
508 mdb_amd64_kvm_framev(void *arglim, uintptr_t pc, uint_t argc, const long *argv,
509     const mdb_tgt_gregset_t *gregs)
510 {
511         /*
```



```
512          * Historically adb limited stack trace argument display to a fixed-
513          * size number of arguments since no symbolic debugging info existed.
514          * On amd64 we can detect the true number of saved arguments so only
515          * respect an arglim of zero; otherwise display the entire argv[].
516          */
517         if (arglim == 0)
518                 argc = 0;

520         mdb_printf("%0?lr %a(", gregs->kregs[KREG_RBP], pc);

522         if (argc != 0) {
523                 mdb_printf("%lr", *argv++);
524                 for (argc--; argc != 0; argc--)
525                         mdb_printf(", %lr", *argv++);
526         }

528         mdb_printf(")\n");
529         return (0);
530 }
```

```
**********************************************************
   12957 Tue Jan 27 15:02:14 2015
new/usr/src/cmd/mdb/intel/mdb/mdb_ia32util.c
5554 kmdb can't trace stacks that begin within itself
Reviewed by: Josef 'Jeff' Sipek <jeffpc@josefsipek.net>
**********************************************************
______unchanged_portion_omitted_

int
mdb_ia32_kvm_stack_iter(mdb_tgt_t *t, const mdb_tgt_gregset_t *gsp,
    mdb_tgt_stack_f *func, void *arg)
{
        mdb_tgt_gregset_t gregs;
        kreg_t *kregs = &gregs.kregs[0];
        int got_pc = (gsp->kregs[KREG_EIP] != 0);
        int err;

        struct fr {
        struct {
                uintptr_t fr_savfp;
                uintptr_t fr_savpc;
                long fr_argv[32];
        } fr;

        uintptr_t fp = gsp->kregs[KREG_EBP];
        uintptr_t pc = gsp->kregs[KREG_EIP];
        uintptr_t lastfp = 0;

        ssize_t size;
        uint_t argc;
        int detect_exception_frames = 0;
        int advance_tortoise = 1;
        uintptr_t tortoise_fp = 0;
#endif /* ! codereview */
#ifndef _KMDB
        int xp;

        if ((mdb_readsym(&xp, sizeof (xp), "xpv_panicking") != -1) && (xp > 0))
                detect_exception_frames = 1;
#endif

        bcopy(gsp, &gregs, sizeof (gregs));

        while (fp != 0) {

                /*
                 * Ensure progress (increasing fp), and prevent
                 * endless loop with the same FP.
                 */
                if (fp <= lastfp) {
                        err = EMDB_STKFRAME;
                        goto badfp;
                }
                if (fp & (STACK_ALIGN - 1)) {
                        err = EMDB_STKALIGN;
                        goto badfp;
                }
                if ((size = mdb_tgt_vread(t, &fr, sizeof (fr), fp)) >=
                    (ssize_t)(2 * sizeof (uintptr_t))) {
                        size -= (ssize_t)(2 * sizeof (uintptr_t));
                        argc = kvm_argcount(t, fr.fr_savpc, size);
                } else {
                        err = EMDB_NOMAP;
                        goto badfp;
                }
```



```
                if (tortoise_fp == 0) {
                        tortoise_fp = fp;
                } else {
                        if (advance_tortoise != 0) {
                                struct fr tfr;

                                if (mdb_tgt_vread(t, &tfr, sizeof (tfr),
                                    tortoise_fp) != sizeof (tfr)) {
                                        err = EMDB_NOMAP;
                                        goto badfp;
                                }

                                tortoise_fp = tfr.fr_savfp;
                        }

                        if (fp == tortoise_fp) {
                                err = EMDB_STKFRAME;
                                goto badfp;
                        }
                }

                advance_tortoise = !advance_tortoise;

#endif /* ! codereview */
                if (got_pc && func(arg, pc, argc, fr.fr_argv, &gregs) != 0)
                        break;

                kregs[KREG_ESP] = kregs[KREG_EBP];

                lastfp = fp;
                fp = fr.fr_savfp;
                /*
                 * The Xen hypervisor marks a stack frame as belonging to
                 * an exception by inverting the bits of the pointer to
                 * that frame.  We attempt to identify these frames by
                 * inverting the pointer and seeing if it is within 0xfff
                 * bytes of the last frame.
                 */
                if (detect_exception_frames)
                        if ((fp != 0) && (fp < lastfp) &&
                            ((lastfp ^ ~fp) < 0xfff))
                                fp = ~fp;

                kregs[KREG_EBP] = fp;
                kregs[KREG_EIP] = pc = fr.fr_savpc;

                got_pc = (pc != 0);
        }

        return (0);

badfp:
        mdb_printf("%p [%s]", fp, mdb_strerror(err));
        return (set_errno(err));
}

/*
 * Determine the return address for the current frame.  Typically this is the
 * fr_savpc value from the current frame, but we also perform some special
 * handling to see if we are stopped on one of the first two instructions of a
 * typical function prologue, in which case %ebp will not be set up yet.
 */
int
mdb_ia32_step_out(mdb_tgt_t *t, uintptr_t *p, kreg_t pc, kreg_t fp, kreg_t sp,
    mdb_instr_t curinstr)
{
```

```
305         struct frame fr;
306         GElf_Sym s;
307         char buf[1];
308
309         enum {
310                 M_PUSHL_EBP     = 0x55, /* pushl %ebp */
311                 M_MOVL_EBP      = 0x8b  /* movl %esp, %ebp */
312         };
313
314         if (mdb_tgt_lookup_by_addr(t, pc, MDB_TGT_SYM_FUZZY,
315             buf, 0, &s, NULL) == 0) {
316                 if (pc == s.st_value && curinstr == M_PUSHL_EBP)
317                         fp = sp - 4;
318                 else if (pc == s.st_value + 1 && curinstr == M_MOVL_EBP)
319                         fp = sp;
320         }
321
322         if (mdb_tgt_vread(t, &fr, sizeof (fr), fp) == sizeof (fr)) {
323                 *p = fr.fr_savpc;
324                 return (0);
325         }
326
327         return (-1); /* errno is set for us */
328 }
329
330 /*
331  * Return the address of the next instruction following a call, or return -1
332  * and set errno to EAGAIN if the target should just single-step.  We perform
333  * a bit of disassembly on the current instruction in order to determine if it
334  * is a call and how many bytes should be skipped, depending on the exact form
335  * of the call instruction that is being used.
336  */
337 int
338 mdb_ia32_next(mdb_tgt_t *t, uintptr_t *p, kreg_t pc, mdb_instr_t curinstr)
339 {
340         uint8_t m;
341
342         enum {
343                 M_CALL_REL = 0xe8, /* call near with relative displacement */
344                 M_CALL_REG = 0xff, /* call near indirect or call far register */
345
346                 M_MODRM_MD = 0xc0, /* mask for Mod/RM byte Mod field */
347                 M_MODRM_OP = 0x38, /* mask for Mod/RM byte opcode field */
348                 M_MODRM_RM = 0x07, /* mask for Mod/RM byte R/M field */
349
350                 M_MD_IND   = 0x00, /* Mod code for [REG] */
351                 M_MD_DSP8  = 0x40, /* Mod code for disp8[REG] */
352                 M_MD_DSP32 = 0x80, /* Mod code for disp32[REG] */
353                 M_MD_REG   = 0xc0, /* Mod code for REG */
354
355                 M_OP_IND   = 0x10, /* Opcode for call near indirect */
356                 M_RM_DSP32 = 0x05  /* R/M code for disp32 */
357         };
358
359         /*
360          * If the opcode is a near call with relative displacement, assume the
361          * displacement is a rel32 from the next instruction.
362          */
363         if (curinstr == M_CALL_REL) {
364                 *p = pc + sizeof (mdb_instr_t) + sizeof (uint32_t);
365                 return (0);
366         }
367
368         /*
369          * If the opcode is a call near indirect or call far register opcode,
370          * read the subsequent Mod/RM byte to perform additional decoding.
```



```
371          */
372         if (curinstr == M_CALL_REG) {
373                 if (mdb_tgt_vread(t, &m, sizeof (m), pc + 1) != sizeof (m))
374                         return (-1); /* errno is set for us */
375
376                 /*
377                  * If the Mod/RM opcode extension indicates a near indirect
378                  * call, then skip the appropriate number of additional
379                  * bytes depending on the addressing form that is used.
380                  */
381                 if ((m & M_MODRM_OP) == M_OP_IND) {
382                         switch (m & M_MODRM_MD) {
383                         case M_MD_DSP8:
384                                 *p = pc + 3; /* skip pr_instr, m, disp8 */
385                                 break;
386                         case M_MD_DSP32:
387                                 *p = pc + 6; /* skip pr_instr, m, disp32 */
388                                 break;
389                         case M_MD_IND:
390                                 if ((m & M_MODRM_RM) == M_RM_DSP32) {
391                                         *p = pc + 6;
392                                         break; /* skip pr_instr, m, disp32 */
393                                 }
394                                 /* FALLTHRU */
395                         case M_MD_REG:
396                                 *p = pc + 2; /* skip pr_instr, m */
397                                 break;
398                         }
399                         return (0);
400                 }
401         }
402
403         return (set_errno(EAGAIN));
404 }
405
406 /*ARGSUSED*/
407 int
408 mdb_ia32_kvm_frame(void *arglim, uintptr_t pc, uint_t argc, const long *argv,
409     const mdb_tgt_gregset_t *gregs)
410 {
411         argc = MIN(argc, (uint_t)arglim);
412         mdb_printf("%a(", pc);
413
414         if (argc != 0) {
415                 mdb_printf("%lr", *argv++);
416                 for (argc--; argc != 0; argc--)
417                         mdb_printf(", %lr", *argv++);
418         }
419
420         mdb_printf(")\n");
421         return (0);
422 }
423
424 int
425 mdb_ia32_kvm_framev(void *arglim, uintptr_t pc, uint_t argc, const long *argv,
426     const mdb_tgt_gregset_t *gregs)
427 {
428         argc = MIN(argc, (uint_t)arglim);
429         mdb_printf("%0?lr %a(", gregs->kregs[KREG_EBP], pc);
430
431         if (argc != 0) {
432                 mdb_printf("%lr", *argv++);
433                 for (argc--; argc != 0; argc--)
434                         mdb_printf(", %lr", *argv++);
435         }
```

```
437         mdb_printf(")\n");
438         return (0);
439 }
```