```
**********************************************************
    1322 Fri Aug 16 20:34:42 2013
new/usr/src/cmd/dis/Makefile
3194 dis crashes disassembling aes
**********************************************************
```

```
   1 #
   2 # CDDL HEADER START
   3 #
   4 # The contents of this file are subject to the terms of the
   5 # Common Development and Distribution License (the "License").
   6 # You may not use this file except in compliance with the License.
   7 #
   8 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9 # or http://www.opensolaris.org/os/licensing.
  10 # See the License for the specific language governing permissions
  11 # and limitations under the License.
  12 #
  13 # When distributing Covered Code, include this CDDL HEADER in each
  14 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15 # If applicable, add the following below this CDDL HEADER, with the
  16 # fields enclosed by brackets "[]" replaced with your own identifying
  17 # information: Portions Copyright [yyyy] [name of copyright owner]
  18 #
  19 # CDDL HEADER END
  20 #
  21 #
  22 # Copyright 2007 Sun Microsystems, Inc.  All rights reserved.
  23 # Use is subject to license terms.
  24 #

  26 PROG=   dis
  27 OBJS=   dis_target.o dis_main.o dis_util.o dis_list.o
  28 SRCS=   $(OBJS:%.o=%.c)

  30 include         ../Makefile.cmd

  32 LDLIBS += -ldisasm -luutil -lelf
  33 CERRWARN += -_gcc=-Wno-uninitialized

  35 .KEEP_STATE:

  37 all: $(PROG)

  39 $(PROG): $(OBJS)
  40         $(LINK.c) -o $@ $(OBJS) $(LDLIBS)
  41         $(POST_PROCESS)

  43 install: all $(ROOTPROG) $(ROOTCCSBINLINK)

  45 clean:
  46         $(RM) $(OBJS) $(PROG)

  48 lint: lint_SRCS

  50 include ../Makefile.targ
  51 include ../Makefile.ctf
  52 #endif /* ! codereview */
```

```
**********************************************************
   23057 Fri Aug 16 20:34:43 2013
new/usr/src/cmd/dis/dis_target.c
3194 dis crashes disassembling aes
**********************************************************
______unchanged_portion_omitted_

712 #if !defined(__sparc)
713 /*
714  * Given an address, return the starting offset of the next symbol in the file.
715  * Only needed on variable length instruction architectures.
716  */
717 off_t
718 dis_tgt_next_symbol(dis_tgt_t *tgt, uint64_t addr)
719 {
720         sym_entry_t *sym;

722         sym = (tgt->dt_symcache != NULL) ? tgt->dt_symcache : tgt->dt_symtab;

724         while (sym != (tgt->dt_symtab + tgt->dt_symcount)) {
722         for (sym = tgt->dt_symcache;
723             sym != tgt->dt_symtab + tgt->dt_symcount;
724             sym++) {
725                 if (sym->se_sym.st_value >= addr)
726                         return (sym->se_sym.st_value - addr);
727                 sym++;
728 #endif /* ! codereview */
729         }

731         return (0);
732 }
733 #endif

735 /*
736  * Iterate over all sections in the target, executing the given callback for
737  * each.
738  */
739 void
740 dis_tgt_section_iter(dis_tgt_t *tgt, section_iter_f func, void *data)
741 {
742         dis_scn_t sdata;
743         Elf_Scn *scn;
744         int idx;

746         for (scn = elf_nextscn(tgt->dt_elf, NULL), idx = 1; scn != NULL;
747             scn = elf_nextscn(tgt->dt_elf, scn), idx++) {

749                 if (gelf_getshdr(scn, &sdata.ds_shdr) == NULL) {
750                         warn("%s: failed to get section %d header",
751                             tgt->dt_filename, idx);
752                         continue;
753                 }

755                 if ((sdata.ds_name = elf_strptr(tgt->dt_elf, tgt->dt_shstrndx,
756                     sdata.ds_shdr.sh_name)) == NULL) {
757                         warn("%s: failed to get section %d name",
758                             tgt->dt_filename, idx);
759                         continue;
760                 }

762                 if ((sdata.ds_data = elf_getdata(scn, NULL)) == NULL) {
763                         warn("%s: failed to get data for section '%s'",
764                             tgt->dt_filename, sdata.ds_name);
765                         continue;
766                 }
```



```
768                 /*
769                  * dis_tgt_section_iter is also used before the section map
770                  * is initialized, so only check when we need to.  If the
771                  * section map is uninitialized, it will return 0 and have
772                  * no net effect.
773                  */
774                 if (sdata.ds_shdr.sh_addr == 0)
775                         sdata.ds_shdr.sh_addr = tgt->dt_shnmap[idx].dm_start;

777                 func(tgt, &sdata, data);
778         }
779 }

781 /*
782  * Return 1 if the given section contains text, 0 otherwise.
783  */
784 int
785 dis_section_istext(dis_scn_t *scn)
786 {
787         return ((scn->ds_shdr.sh_type == SHT_PROGBITS) &&
788             (scn->ds_shdr.sh_flags == (SHF_ALLOC | SHF_EXECINSTR)));
789 }

791 /*
792  * Return a pointer to the section data.
793  */
794 void *
795 dis_section_data(dis_scn_t *scn)
796 {
797         return (scn->ds_data->d_buf);
798 }

800 /*
801  * Return the size of the section data.
802  */
803 size_t
804 dis_section_size(dis_scn_t *scn)
805 {
806         return (scn->ds_data->d_size);
807 }

809 /*
810  * Return the address for the given section.
811  */
812 uint64_t
813 dis_section_addr(dis_scn_t *scn)
814 {
815         return (scn->ds_shdr.sh_addr);
816 }

818 /*
819  * Return the name of the current section.
820  */
821 const char *
822 dis_section_name(dis_scn_t *scn)
823 {
824         return (scn->ds_name);
825 }

827 /*
828  * Create an allocated copy of the given section
829  */
830 dis_scn_t *
831 dis_section_copy(dis_scn_t *scn)
832 {
833         dis_scn_t *new;
```

```
835         new = safe_malloc(sizeof (dis_scn_t));
836         (void) memcpy(new, scn, sizeof (dis_scn_t));

838         return (new);
839 }

841 /*
842  * Free section memory
843  */
844 void
845 dis_section_free(dis_scn_t *scn)
846 {
847         free(scn);
848 }

850 /*
851  * Iterate over all functions in the target, executing the given callback for
852  * each one.
853  */
854 void
855 dis_tgt_function_iter(dis_tgt_t *tgt, function_iter_f func, void *data)
856 {
857         int i;
858         sym_entry_t *sym;
859         dis_func_t df;
860         Elf_Scn *scn;
861         GElf_Shdr       shdr;

863         for (i = 0, sym = tgt->dt_symtab; i < tgt->dt_symcount; i++, sym++) {

865                 /* ignore non-functions */
866                 if ((GELF_ST_TYPE(sym->se_sym.st_info) != STT_FUNC) ||
867                     (sym->se_name == NULL) ||
868                     (sym->se_sym.st_size == 0) ||
869                     (sym->se_shndx >= SHN_LORESERVE))
870                         continue;

872                 /* get the ELF data associated with this function */
873                 if ((scn = elf_getscn(tgt->dt_elf, sym->se_shndx)) == NULL ||
874                     gelf_getshdr(scn, &shdr) == NULL ||
875                     (df.df_data = elf_getdata(scn, NULL)) == NULL ||
876                     df.df_data->d_size == 0) {
877                         warn("%s: failed to read section %d",
878                             tgt->dt_filename, sym->se_shndx);
879                         continue;
880                 }

882                 if (tgt->dt_shnmap[sym->se_shndx].dm_mapped)
883                         shdr.sh_addr = tgt->dt_shnmap[sym->se_shndx].dm_start;

885                 /*
886                  * Verify that the address lies within the section that we think
887                  * it does.
888                  */
889                 if (sym->se_sym.st_value < shdr.sh_addr ||
890                     (sym->se_sym.st_value + sym->se_sym.st_size) >
891                     (shdr.sh_addr + shdr.sh_size)) {
892                         warn("%s: bad section %d for address %p",
893                             tgt->dt_filename, sym->se_sym.st_shndx,
894                             sym->se_sym.st_value);
895                         continue;
896                 }

898                 df.df_sym = sym;
899                 df.df_offset = sym->se_sym.st_value - shdr.sh_addr;
```



```
901                 func(tgt, &df, data);
902         }
903 }

905 /*
906  * Return the data associated with a given function.
907  */
908 void *
909 dis_function_data(dis_func_t *func)
910 {
911         return ((char *)func->df_data->d_buf + func->df_offset);
912 }

914 /*
915  * Return the size of a function.
916  */
917 size_t
918 dis_function_size(dis_func_t *func)
919 {
920         return (func->df_sym->se_sym.st_size);
921 }

923 /*
924  * Return the address of a function.
925  */
926 uint64_t
927 dis_function_addr(dis_func_t *func)
928 {
929         return (func->df_sym->se_sym.st_value);
930 }

932 /*
933  * Return the name of the function
934  */
935 const char *
936 dis_function_name(dis_func_t *func)
937 {
938         return (func->df_sym->se_name);
939 }

941 /*
942  * Return a copy of a function.
943  */
944 dis_func_t *
945 dis_function_copy(dis_func_t *func)
946 {
947         dis_func_t *new;

949         new = safe_malloc(sizeof (dis_func_t));
950         (void) memcpy(new, func, sizeof (dis_func_t));

952         return (new);
953 }

955 /*
956  * Free function memory
957  */
958 void
959 dis_function_free(dis_func_t *func)
960 {
961         free(func);
962 }
```